TACKLING BUILD FAILURES IN CONTINUOUS INTEGRATION

by

FOYZUL HASSAN, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Xiaoyin Wang, Ph.D., Chair
Jianwei Niu, Ph.D.
Wei Wang, Ph.D.
Palden Lama, Ph.D.
Lingming Zhang, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2020

DEDICATION

To my parents
To my wife Fariha
To rest of my family

ACKNOWLEDGEMENTS

First of all, I would like to thank my adviser, Dr. Xiaoyin Wang. Without his guidance, support, and assistance, it would not be possible for me to finish my dissertation. I very much enjoyed spending time and working with him. Dr. Wang always helped in identifying, formulating, and solving research problems, and patiently advised the publications and presentations to improve the quality of my research results. Apart from research work, his advises on professional career paths and preparation for professional carrier have been invaluable. Besides my advisor, I would like to thank the rest of my dissertation committee: Prof. Jianwei Niu, Prof. Palden Lama, Prof. Wei Wang and Prof. Lingming Zhang for their support, insightful comments, constructive feedback, and challenging questions. I am thankful to Prof. Na Meng from Virginia Tech for being a great collaborator and imparting her knowledge of fault-localization techniques. I am also grateful to my research internship mentors at Microsoft Research Nachi Nagappan, Tom Zimmermann, Chetan Bansal and Ahmed Hassan Awadallah for their guidance and fruitful discussion on identifying new research problems.

I am privileged to be associated with the current software engineering group at UTSA and like to thank my fellow lab mates: Shaikh Nahid Mostafa, and Xue Qin. Without your great and enjoyable company, it was not possible for me to pass the long journey here.

My sincerest gratitude and appreciation go to my lovely wife Fariha Nusrat for constantly supporting me and going through hardships during my PhD years. I would also like to thank my parents, my brother and my sister for their never-ending love, care, and support.

Finally, I would like to thank the financial support provided by the Department of Computer Science at UTSA, as well as the funding support from the National Science Foundation (Grant:1464425,1846467).

This Masters Thesis/Recital Document or Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Masters Thesis/Recital Document or Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Masters Thesis/Recital Document or Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Masters Thesis/Recital Document or Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this Masters Thesis/Recital Document or Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Masters Thesis/Recital Document or Doctoral Dissertation attest to the accuracy of this statement.

May 2020

TACKLING BUILD FAILURES IN CONTINUOUS INTEGRATION

Foyzul Hassan, Ph.D.
The University of Texas at San Antonio, 2020

Supervising Professor: Xiaoyin Wang, Ph.D.

In popular continuous integration(CI) practice, coding is followed by building, integration and system testing, pre-release inspection and deploying artifacts. This can reduce integration time and speed up the development process. At CI environment, dedicated infrastructure with different build systems such as Make, Ant, Maven, Gradle, Bazel, etc. are used to automate CI tasks like compilation, test invocation, etc. For continuous integration, developers describe build process though build scripts such as build.xml for Ant, pom.xml for Maven, and build.gradle for Gradle. But with the growing functionality and CI requirement, build scripts can very complex and may require frequent maintenance. Meanwhile, a large number of continuous integration failures may interrupt normal software development so that they need to be repaired as soon as possible. According to the TravisTorrent dataset of open-source software continuous integration, 22% of code commits include changes in build script files to maintain build scripts or to resolve CI failures.

CI failures bring both challenges and opportunities to fault localization and program repair techniques. On the challenge side, unlike traditional fault localization scenarios (e.g., unit testing, regression testing) where only source code needs to be considered, CI failures can also be caused by build configuration errors and environment changes. Actually, the CI practice has make it a necessity for developers to automate the software build process with build scripts and configuration files. As a result, the build maintenance needs a substantial amount of developer efforts, and developer's carelessness may lead to defects in build scripts also.

On the opportunity side, the scenario of continuous integration provides rich code commit history and build logs from previous passing builds and current failing builds. Such information sources are often not available in other application scenarios. Taking advantage of these additional

information sources, we may be able to further improve the accuracy of automatic repair and fault localization for CI failures.

Automated program repair techniques have great potential to reduce the cost of resolving software failures, but the existing techniques mostly focus on repairing source code so that they cannot directly help resolving software CI failures. To address this limitation, we proposed the first approach to automatic patch generation for build scripts, using fix patterns automatically generated from existing build script fixes and recommending fix patterns based on build log similarity.

Apart from build script, CI failures are often a combination of test failures and build failures, and sometimes both source code and various build scripts need to be touched in the repair of one failure. To address this issue, we proposed a unified technique to localize faults in both source code and build scripts given CI failures. 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 naively applying an off-the-shelf IR technique to these software artifacts, for more accurate fault localization, UniLoc (Unified Fault Localization) applies various domain-specific heuristics to optimize the search queries, search space, and ranking formulas. However, UniLoc can localize faults up to file level and limited to find faults in source code and build scripts. In the future, 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. Proposed CI fault localization technique with program repair can be a handy solution to fix a range of CI failures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Motivation

Due to the ever-increasing nature of software requirements and rigorous release practices, Continuous Integration (CI) [80] is gaining more and more popularity. In CI environment, developers merge code in a codebase, followed by automatic software build process execution, test execution and artifact deployment. This practice allows developers to detect more software faults earlier and also request developers to resolve the detected fault in a timely manner to make the release pipeline flawless. A study on Google continuous integration [196] finds that more than 5,500 code commits are merged to the codebase and 100 million test cases are executed per day for validation.

At CI environment, dedicated infrastructure with different build systems such as Make, Ant, Maven, Gradle, Bazel, etc. are used to automate CI tasks like compilation, test invocation, etc. For continuous integration, developers describe build process though build scripts such as build.xml for Ant, pom.xml for Maven, and build.gradle for Gradle. But with the growing functionality and CI requirement, build scripts can very complex and may require frequent maintenance [140]. Our study [99] on TravisTorrent dataset [62] finds that around 29% of CI trials fail. Seo et al. [178] also addressed a similar issue with 37% build failure proportion at Google CI environment. Rausch et al. [172] find that dependency resolution, compilation, and configuration phases account for 22% of CI failures. On the other hand, 65% failures are categorized as test failures and 13% as quality checking errors. Our study on 1,187 CI failures on TravisTorrent Dataset also confirms that 10.8% of CI-failure fix contains only build script revisions, and 25.6% CI-failure contains both build script and source code.

Apart from CI failure issues, the integration process might be delayed for long build chains, build error fixes, and frequent code changes in version control system. According to the analysis on TravisTorrent [62] data, the median build time for Java project is over 900 seconds and the median length of continuous build failure sequences is 4. Thus when multiple developers are committing their changes concurrently, their commits may be piled up in the building queue, and they may

need to wait for a long time to get the build feedback. They also have the option to continue working on their copy, but they will be at the risk of rolling back their changes if their original commit fails the integration.

CI failures bring both challenges and opportunities to program repair and fault localization techniques. On the challenge side, unlike traditional fault localization scenarios (e.g., unit testing, regression testing) where only source code needs to be considered, CI failures can also be caused by build configuration errors and environment changes. Actually, the CI practice has made it a necessity for developers to automate the software build process with build scripts and configuration files. As a result, the build maintenance needs a substantial amount of developer efforts, and developers' carelessness may lead to defects in build scripts also. This has been confirmed by various empirical studies. The study by Kumfert et al. [113] found that build maintenance may impose 12% to 36% overhead on software development, and the study by McIntosh et al. [140] shows that, on average, build scripts account for 9% of all maintained files, and 20% of code commits involve build scripts. Furthermore, although it is possible to differentiate test failures and build failures, it is often not possible to ascribe the failure to source code or build scripts just based on the failure type. On the opportunity side, the scenario of continuous integration provides rich code commit history and build logs from previous passing builds and current failing builds. Such information sources are often not available in other application scenarios of program repair and fault localization. Taking advantage of these additional information sources, we may be able to further improve the accuracy of program repair and fault localization for CI build failures. Our proposed method focuses on fault localization and program in a CI environment.

Apart from build repair and fault localization, it is desirable to have a recommendation system that predicts the build feedback of a code commit and thus gives developers more confidence to continue their work and reduce the chance of rolling back. With our proposed approach, we analyzed software build execution time, commit and consecutive build status change to study the necessity and possibility of a change-aware build prediction model. Furthermore, we propose a recommendation system to predict build outcome based on the TravisTorrent data and also the

code change information in the code commit such as import statement changes, method signature changes to train the build prediction model.

For faster deployment and integration, OS-level container such as Docker is heavily used for continuous deployment (CD). A container image is a stand-alone and executable package of a piece of software with all its environment dependencies, including code, runtime, system tools, libraries, file structures, settings, etc. Despite the large benefit brought by container images during software deployment, they also increase the effort of software developers because they need to generate and maintain the image configuration files which describe how the container images can be constructed with all environment dependencies, such as what tools and libraries should be installed and how the file structure should be set up. Our work on Dockerfile can help developers to update container image configuration files more easily and with more confidence. We use novel string analysis for DockerFile and source code and performed dependency analysis for environment dependency mapping. Finally, to generate DockerFile update suggestion, we used Change Impact Analysis to provide a suggestion based on code changes. Our evaluation on 1,199 real-world instruction updates shows that RUDSEA can recommend correct update locations for 78.5% of the updates, and correct code changes for 44.1% of the updates. In the future, we planned to utilize system trace information such as Ptrace, Strace, etc. to our earlier developed fault localization research work and develop a complete tool suite to repair build failures involving heterogeneous failure types and files.

1.2 Thesis Statement

Our thesis is five-pronged:

> ***(1)*** *Required to have a detailed study on software build failures and approach to resolve these build failures.*
> ***(2)*** *There should have a build prediction model that can detect build outcome based on code change and build failure type*
> ***(3)*** *There is a need of build repair technique to resolve build failures.*
> ***(4)*** *In need to have an approach to localize faults that involves both source code and build scripts.*
> ***(5)*** *Required to have tool support for deployment environment analysis to avoid integration problems.*

1.3 Contributions

To confirm the thesis statement, this dissertation makes the following contributions:

- The dissertation presents a feasibility study on automatic software building. Particularly, we first put state-of-the-art build automation tools (Ant, Maven and Gradle) to the test by automatically executing their respective default build commands on top 200 Java projects from GitHub. Next, we focus on the 86 projects that failed this initial automated build attempt, manually examining and determining correct build sequences to build each of these projects.

- To avoid a stall in CI pipeline, we proposed build prediction model that uses TravisTorrent data set with build error log clustering and AST level code change modification data to predict whether a build will be successful or not without attempting actual build so that the developer can get early build outcome results. With the proposed model, we can predict

build outcome with an average F-Measure over 87% on all three build systems (Ant, Maven, Gradle) under the cross-project prediction scenario.

- The dissertation presents an approach HireBuild: History-Driven Repair of Build Scripts, the first approach to automatic patch generation for build scripts, using fix patterns automatically generated from existing build script fixes and recommending fix patterns based on build log similarity. From TravisTorrent dataset, we extracted 175 build failures and their corresponding fixes, which revise Gradle build scripts. Among these 175 build failures, we used the 135 earlier build fixes for automatic fix-pattern generation and the more recent 40 build failures (fixes) for evaluation of our approach. Our experiment shows that our approach can fix 11 of 24 reproducible build failures, or 45% of the reproducible build failures, within a comparable time of manual fixes.

- Proposed a unified technique to localize faults in *both* source code *and* build scripts given CI failures. 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 natively 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 63 open source projects that are built with Gradle. UniLoc could effectively locate bugs with the average MRR (Mean Reciprocal Rank) value as 0.49 and MAP (Mean Average Precision) value as 0.36. UniLoc outperformed the state-of-the-art IR-based tool BLUiR, whose MRR and MAP values were 0.29 and 0.19. UniLoc has the potential to help developers diagnose root causes for CI failures more accurately and efficiently.

- Recommendation system of dockerfiles for developers based on analyzing changes in software environment assumptions and their impacts. Our evaluation on 1,199 real-world instruction updates shows that RUDSEA can recommend correct update locations for 78.5%

of the updates, and correct code changes for 44.1% of the updates.

1.4 Organization

This dissertation is organized as follows. Chapter 2 introduces the background and related work. Chapter 3 describes our empirical study on the software build failures and approaches to resolve the build failures automatically. Chapter 4 presents a build prediction model to avoid a stall in CI pipeline. Chapter 5 and chapter 6 presents our automatic program repair approach to fix build script issues and unified fault localization technique for source code and build script. While in chapter 7 we discussed our work on docker environment analysis. In chapter 8, we discussed the conclusion. Finally, chapter 9 presents an overview of our future work on localization and repair of test and build failure.

CHAPTER 2: BACKGROUND AND RELATED WORK

The purpose of this section is to provide the background of this study and a review of related literature. First, the background is introduced, followed by a related work section about Analysis of Build Configuration Files, Automatic Program Repair, and Fault Localization.

2.1 Analysis of Build Configuration Files

Analysis of build configuration files is growing as an important aspect for software engineering research such as dependency analysis for path expression, migration of build configuration file and empirical studies. On dependency analysis, Gunter [51] proposed a Petri-net based model to describe the dependencies in build configuration files. Adams et al. [46] proposed a framework to extract a dependency graph for build configuration files, and provide automatic tools to keep consistency during revision. Most recently, Al-Kofahi et al. [48] proposed a fault localization approach for make files, which provides the suspiciousness scores of each statement in a make files for a building error. Wolf et al. proposed an approach [215] to predict build errors from the social relationship among developers. McIntosh et al. [140] carried out an empirical study on the efforts developers spend on the building configurations of projects. Downs et al. [79] proposed an approach to remind developers in a development team about the building status of the project. On the study of building errors, Seo et al. [178] and Hassan et al. [94, 97] carried out empirical studies to categorize build errors. Their study shows that missing types and incompatibility are the most common type of build errors, which are consistent with our findings.

The most closely related work in this category is SYMake developed by Tamrawi et al. [188]. SYMake uses a symbolic-evaluation-based technique to generate a string dependency graph for the string variables/constants in a Makefile, automatically traces these values in maintenance tasks (e.g.,renaming), and detect common errors. Compared to SYMake, the proposed project plans to develop build configuration analysis for a different purpose (i.e., automatic software building). Therefore, the proposed analysis estimates run-time values of string variables with grammar-based

string analysis instead of string dependency analysis and analyzes flows of files to identify the paths to put downloaded files and source files to be involved. On migration of build configuration files, AutoConf [12] is a GNU software that automatically generates configuration scripts based on detected features of a computer system. AutoConf detects existing features (e.g., libraries, software installed) in a build environment, and configure the software based on pre-defined options.

2.1.1 Automatic Program Repair

Automatic program repair is gaining research interest in the software engineering community with the focus to reduce bug fixing time and effort. Recent advancements in program analysis, synthesis, and machine learning have made automatic program repair a promising direction. Early software repair techniques are mainly specific to predefined problems [111, 181, 204, 231]. Le Goues et al. [89] GenProg, which is one of the earliest and promising search-based automatic patch generation technique based on genetic programming. Patch generated by this approach follows random mutation and use test case for the verification of the patch. Later in 2012, authors optimized their mutation operation and performed systematic evaluation 105 real bugs [89]. RSRepair [164] performs similar patch generation based on a random search. D. Kim et al. [112] proposed an approach to automatic software patching by learning from common bug-fixing patterns in software version history, and later studied the usefulness of generated patches [191]. AE [209] uses a deterministic search technique to generate patch. Pattern-based Automatic Program Repair(PAR) [112] uses manually generated templates learned from human-written patches to prepare a patch. PAR also used a randomized technique to apply the fix patches. Nguyen et al. [152] proposed SemFix, which applied software synthesis to automatic program repair, by checking whether a suspicious statement can be re-written to make a failed test case pass. Le et al. [120] mines bug fix patterns for automatic template generation and uses version control history to perform mutation. Prophet [128] proposed a probabilistic model learned from human-written patches to generate a new patch. The above-mentioned approaches infer a hypothesis that a new patch can be constructed based on existing sources. This hypothesis also validated by Barr et al. [56] that 43 percent changes can be

generated from existing code. With this hypothesis, we proposed the first approach for automatic build failure patch generation. Tan and Roychoudhury proposed Relifix [189], a technique that is taking advantage of version history information to repair regression faults. Smith et al. [183] reported an empirical study on the overfitting to test suites of automatically generated software patches. Most recently, Long and Rinard proposed SPR [127], which generates patching rules with condition synthesis, and searches for the valid patch in the patch candidates generated with the rules. Angelix [144] and DirectFix [143] both use semantics-based approach for patch generation. To fix buggy conditions, Nopol [220] proposes a test-suite based repair technique using Satisfiability Modulo Theory(SMT). Although our fundamental goal is same, our approach is different than others in several aspects: 1) Our approach is applicable for build scripts, 2) We generate automatic fix template using build failure log similarity, 3) With abstract fix template matching we can generate fix candidate lists with reasonable size.

2.2 Automatic Fault Localization

Automatic bug localization has been active research over the decades [125] [236]. Automatic bug localization techniques can be broadly divided into two categories: i) Dynamic and ii) Static. Dynamic fault localization [159] requires the execution of program and test cases to identify precise fault location. As a result, dynamic fault localization techniques can be expensive and might require an execution environment. Among the dynamic fault localization techniques, spectrum-based fault localization(SBFL) [45] is the most prominent technique. SBFL usually depends on the suspicion score based on program elements executed by the test cases. Tarantual [108] is the early research work on SBFL and subsequent researchers are working to improve the accuracy of localization techniques. Ochiai [44] uses different similarity co-efficient to find more accurate fault localization. Xuan and Monperrius proposed Multric [221], which combines learning-to-rank and fault localization techniques for more accurate localization. Recent work on fault localization: Savant [54] uses like invariant with a learning-to-rank algorithm for fault localization. Since software building process lacks test cases and takes time to build software, dynamic approaches might

be overhead for CI environment.

As opposed to the dynamic fault localization approach, static fault localization techniques do not require test cases and execution information. Static fault localization depends on static source code analysis [233] [78] or information retrieval based approaches [200] [236]. Lint [107] is one of first tool to find fault in `C` programs. FindBug [53] and PMD [26] are prominent static code analyzer for Java source code. Lots of IR-based approaches [236] [177] have been proposed by the researchers for fault localization. BugLocator [236] performs bug localization based on the revised VSM model. Saha et al. [177] proposed BLUiR considering source code structure for IR-based fault localization. Recent work on fault localization Locus [211] utilizes change history for fault localization. Since static fault localization does not require execution environment and test cases, we also 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 files.

2.3 Containerized Software Development

Modern software often depends on a large variety of environment dependencies to be properly deployed and operated on production machines. Databases, application servers, system tools, and supporting files often need to be well installed and configured before software execution, and thus may cause tremendous effort and high risks during software deployment. A practical approach to alleviate this effort is to use container images. A container image [104] is a stand-alone and executable package of a piece of software with all its environment dependencies, including code, runtime, system tools, libraries, file structures, settings, etc. It can be easily ported and deployed to other machines but is much lighter-weight than traditional virtual machines that can achieve similar goals. Container-based virtualization systems such as Kubernetes [23], Docker [9], etc. are widely used in software development process for faster deployment and reproduction. Recent research work [71] on Continuous Development(CD) identifies that CD workflow or pipeline widely depends on containerized images. Despite the large benefit brought by container images during

software deployment, they also increase the effort of software developers because they need to generate and maintain the image configuration files.

CHAPTER 3: AUTOMATIC BUILDING OF JAVA PROJECTS IN SOFTWARE REPOSITORIES: A STUDY ON FEASIBILITY AND CHALLENGES

In this chapter, we present a feasibility study on automatic software building. Particularly, we first put state-of-the-art build automation tools (Ant, Maven and Gradle) to the test by automatically executing their respective default build commands on top Java projects from GitHub. A significant portion of this work has been presented at the following venue [94]:

- F. Hassan, S. Mostafa, E. S. L. Lam and X. Wang, "Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, 2017, pp. 38-47.

3.1 Introduction

Over the past decade, open software repositories such as GitHub [67], Sourceforge [31] and Google Code [15] are gaining popularity: the number of publicly available repositories have increased tremendously, as software developers are starting to exploit the power of communal open-source development, from small-scale pet projects, to large-scale industrial strength applications (e.g., Android). The availability of open software repositories have presented the software engineering and research communities with a unique opportunity: we now have access to a large corpus of source code from a wide range of software applications that collectively contain immensely rich data. Even at the time of writing this paper, a large number of software engineering techniques have been developed to analyze and mine data (e.g., code, version history, bug reports) from public software repositories [93] [148] [202]. While meta-data of such projects are important, the most significant amount of data is hidden in the syntax and semantics of the code base. Hence, it is often necessary to apply program analysis [190] [231] techniques to these repositories. To do so

at a large-scale, it is important that we develop techniques to automate every step of the analysis pipeline, reducing or even eliminating the need for human intervention.

Automating this analysis pipeline is very challenging in general. Best practices dictate that open software repositories store and maintain only source code of software projects, with consensus that outputs (e.g., bytecode and binaries) are expected to built in the local development environments. While some program analysis techniques (e.g., PPA [74]) can be applied to just the source code of the repositories, many useful analysis techniques (e.g., points-to analysis [123], call-graph generation [126], string analysis [124]) depend on either the resolution of dependencies, types, and bindings to extract dependable information, or the build artifacts (bytecode, program traces) as input of the analysis. Furthermore, many well-maintained frameworks (e.g., Soot [116], Wala [19], and LLVM [118]) require built software or a successful build process, and a large number of program analysis techniques are based on these frameworks. Thus, automatic building of project repositories is without doubt, a crucial part of this analysis pipeline.

Yet, in spite of the availability of powerful build automation and integration tools like Ant, Maven and Gradle, the task of building public software repositories is often not entirely an automated endeavor in practice. In this paper, we perform a feasibility study on automatically building software projects downloaded from open software repositories. Specifically, we downloaded the most popular 200 Java projects (by number of stars) from GitHub [67], and applied to each project the default execution command of three popular state-of-the-art Java building tools: Maven [145], Ant [192] and Gradle [105]. We chose GitHub as the source of projects because it is most popular hosting website for open source projects, and it provides a standard interface for meta-data and source code downloading. At GitHub, project users can give stars to a project to show their endorsement, and the number of stars, to a large extent, reflect the size of a project's user group and its popularity among users.

For each project with build failures, we manually examined and uncovered the root causes of the build failures, and manually implemented the fixes to the build scripts where possible. Following this, we categorized all the root causes of failures, and studied whether and how each failure

category can be automatically identified and fixed. Our study focuses on Java projects, particularly because it is a mature and widely adopted platform-independent programming language. We also focus on extracting builds from build configuration tools: Ant, Maven and Gradle as they are the three most popular build tools and cover most of open-source Java projects as identified in TravisTorrent [62] data set.

We wish also to emphasize that since detailed build instructions are not always available in open source projects (as shown in our study result in Table 3.2), most users of these top Java projects and other similar projects in GitHub will typically resort to using default build commands, as we did, in the attempt to build these popular projects. Hence, it is very likely that they will face exactly the same build failures that we have curated here. Therefore, our study is not only useful for automatic software building tools and software engineering researches, but also helpful for project developers and build-tool designers to refine their work and reduce build failures faced by software users.

From our study, we have the following five major findings.

- Gradle and Maven are the two dominating building tools for top Java projects. They are used in 174 of top 200 Java projects.

- 99 of 200 top Java projects cannot be built successfully with default build commands, among which 2 do not have source code, 11 are not using Maven, Ant or Gradle as their building tools, and the remaining 86 projects have various build failures.

- In our hierarchical taxonomy on the root causes of 91 detected build failures (from 86 projects), the leading categories are backward-incompatibility of JDK and building tools, non-default parameters in build commands, and project defects in code / configuration files, which accounts of 19, 33, and 14 build failures, respectively.

- Among 91 build failures, 12 have information in the project's readme file that guides their resolution, and 27 have information in the build failure log that guides the identification of their root causes.

- Among 91 build failures, at least 52 can be automatically resolved by extracting/predicting correct build commands from readme files, exhaustive trial of JDK/build-tool versions, and generation of dummy files.

3.2 Study Design

3.2.1 Research Questions

In our feasibility study, we expect to answer the following research questions.

- **RQ1:** What proportion of top Java projects can be successfully built with default build commands of popular build tools?
- **RQ2:** What are the major root causes of the observed build failures?
- **RQ3:** How easily can root causes of build failures be identified from readme files and build failure logs?
- **RQ4:** What proportion of build failures can be (or have the potential to be) automatically resolved?

3.2.2 Study Setup

To perform our study, we downloaded top 200 Java projects from GitHub ranked by the number of stars. We used popularity as project selection criteria since popular projects are more likely to be selected for large-scale software analysis and studies. The downloading was performed as cloning the latest commit as of Aug 30, 2016. Building of software projects not only depends on build commands, but also depends on the build environment (e.g., Java Compiler, build tools). Our build environment includes Ubuntu 14.10, Java SDK 1.8.0_65, Android SDK 24.4.1, Maven 3.3.9, Gradle 3.1, and Ant 1.9.3. We also set environment variables for Java, Android, Ant, Maven and Gradle runtime environments according to their installation guides. When investigating the build failures, we made the necessary customization to the build environment (e.g., reverting to required

version of JDK or Android SDK) in order to resolve the respective build issues and achieve a successful build.

3.3 Study on Successfulness of Default Build Commands (RQ1)

To answer **RQ1**, we developed a systematic way to automatically apply the default build commands to each project and determine the outcome of the build attempt. To determine which default build command to use for each specific build tool, our study on build instructions in readme files [8] derived the most frequently used commands, and hence offering the most likelihood of success: for Maven (`mvn compile`), Ant (`ant build`), and Gradle (`./gradlew` and `gradle build` for projects with and without wrappers respectively). A straightforward way of applying build commands is to run them in the root folder. However, we found that among 200 Java projects for study only 35 projects contain build configuration file directly in the root directory. Therefore, we use the following systematic strategy to determine in which folder we apply the default build commands. We first identify a set of folders F that directly contain a build configuration file (i.e., `pom.xml` for Maven, `build.xml` for Ant, `build.gradle` or `gradlew.bat` for Gradle.). If a folder f in F is not directly (or transitively) contained by any other folders in F, we choose f as the folder to apply build commands. If there are multiple folders satisfying this condition, which indicates that multiple sub-projects may be built independently, we choose the folder that transitively contains most Java source files to apply build commands. In case multiple build files present in the folder, we use the file corresponding to the newer building system, as the other files are more likely to be legacy files for maintenance and in many cases older build systems are outdated to build the project. Specifically, Gradle has priority over Maven and Maven has priority oven Ant. Finally, we determine a build to be successful if both two conditions are hold: 1) the exit code of the build process is 0 and 2) no build failure messages are in the build log .

The results of applying default build commands to top 200 projects are presented in Table 3.1. From the table, we have the following observations. First, Maven and Gradle are the dominating build tools used in top Java projects from GitHub. They collectively encompass more than 85%

Table 3.1: Overall Result of Executing Default Build Commands

Build Tool	Maven	Gradle	Ant	Other	No Source Code	All
# Projects Built Successful	40	53	8	N/A	N/A	101
# Projects Failed	25	56	5	N/A	N/A	86
# Total Projects	65	109	13	11	2	200

of the projects, hence for automatic software building tools, it is reasonable to focus on only these popular tools. This is also the reason we focus on the build failures of popular build tools in our study. Second, 86 (46%) of 187 projects using Maven, Ant and Gradle build tools failed in the building process. This low successful-build rate essentially shows the necessity of more advanced features for state-of-the-art build automation tools. Interestingly, since the top projects are typically well maintained, we can naturally expect build rates to decline in a sample of more average projects in GitHub. Third, Maven has a higher build successful rate than Gradle. While they share similar design ideas, we suspect that Gradle fails in more projects because it allows more customization, and is widely used in Android projects which are more complicated than normal Java projects due to configuration description for Android SDK, NDK and device information. Figure 3.2 partly verified our guess. Finally, we found 13 projects in our top 200 samples containing no Ant, Maven or Gradle build scripts. Of these, 2 projects have no source code, suggesting that some form of filtering is necessary when processing projects from open software repositories. The other 11 projects contain customized build scripts written by the developers. While it is still possible to analyze and automatically build these project, the lack of standardization and variety of building mechanisms are likely to pose significant challenges.

3.4 A Taxonomy of Root Causes of Build Failures (RQ2)

To answer RQ2, we categorized all the build failures from the 86 projects that failed to be built with default build commands. To clearly confirm the root cause of the build failures, we manually examined and resolved the respective build issues, until we successfully built the project. During this iterative process, we discovered 5 more build failures that were not identified during the builds with default build commands. They are reported after we fixed the first-seen build failures. The

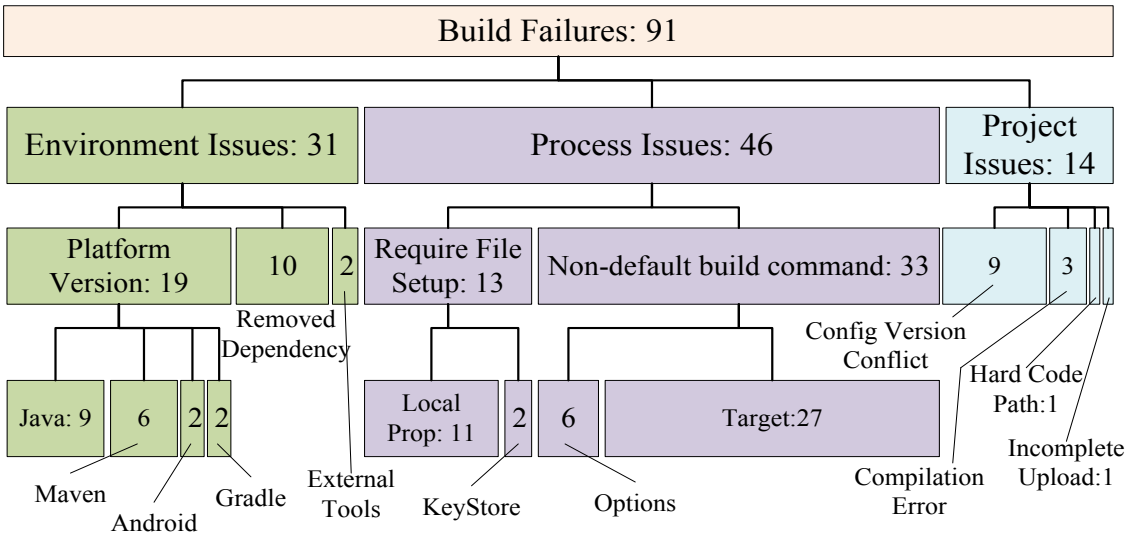


Figure 3.1: Build Failure Hierarchy

reason is that, build tools (actually also compilers) typically stop when they face a fatal error. To make sure we are not biased to the first-seen build failures (which are typically in the earlier stage of building process), we include these 5 build failures into our categorization study.

Our taxonomy generated by the agreement of the first author and the fourth author on build failures by their root causes are presented in Figure 3.1. In the figure, each colored block represents a category (or sub-category), and the blocks' widths indicate the number of build failures in the category. Finally, higher level categories are linked to their direct sub-categories with elbow lines. We classify the build failures to 3 general categories: environment issues, process issues, and project issues. We will detail these categories and their subcategories in the following subsections.

3.4.1 Environment Issues

Environment issues are build failures caused by the change of building environment. They happen when a necessary component in the local system or a remote server becomes unavailable. 31 of the 91 build failures we found are environment issues, and they fall into 3 sub-categories: platform version issues, removed dependency, and requirement of external tools.

Platform Version Issues: This is the largest sub-category of environment issues with 19 of 31 issues fall into it. A platform version issue happens when the successful building of the project

relies on a specific historical version of build tools or SDK. Interestingly, though many of these projects exhibited recent code changes, their respective developers have chosen to use older SDK or Build Tools. The reason can be either some dependencies on a specific SDK or build tool version, or developers did not feel that the upgrade was needed (or worth his/her time). An example of platform version issue is shown as BuildLogExample 1. In this example and all following examples, we put the project name (repository name / project name) and commit number at the right corner of the example title. In this example, elasticSearch uses a Gradle feature that became unavailable after version 2.14, so the project can only be built with Gradle 2.13.

BuildLogExample 1 Platform Version Issue 1 *(elasticSearch/elasticSearch: 42a7a55)*

```
A problem occurred evaluating root project 'buildSrc'.
      > Gradle 2.13 is required to build elasticsearch
```

Actually, Build failure logs do not always reveal the root cause of the failures. For example, Example 14 is another build failure example from Facebook Rebound, which shows an error when running Java in the `rebound-android-example:preDexDebug` task of gradle script, but it is very difficult to tell that this error is due to a backward incompatibility of Java 8.

We further studied which build tool or SDK are causing platform version issues. Among the 19 environment issues, 9 requires Java 7 SDK, so we need to downgrade Java from 8 to 7 to solve them; 6 require older Maven versions; 2 require older Gradle versions; and 2 require older Android SDK versions.

BuildLogExample 2 Platform Version Issues 2 *(facebook/rebound: 5017fc9)*

```
* What went wrong:
Execution fail for task ':rebound-android-example:preDexDebug'.
> com.android.ide.common.process.ProcessException: org.gradle.process.
   internal.ExecException: Process 'command '/usr/local/java/jdk1.8.0
   _65/bin/java'' finished with non-zero exit value 1
```

Removed Dependency: In Maven and Gradle, dependency Jar files are stored at central repository or developer-specified dependency repositories. Central repositories manage archive files to store old versions of Jar files, but in many cases, such archives can be removed from the reposi-

tory. This removal will unfortunately break the build scripts of any projects relying on them. 9 out of total 10 issues on the removal of dependencies are caused by a certain Jar file being removed from the maven central repository. In Example 3, `com.android.tools.build/gradle 2.0.0-alpha1` no longer exist in the central repository of Maven (perhaps been replaced with a more stable version because the version 2.0.0 does exist).

BuildLogExample 3 Removal of Dependency *(Yalantis/Phoenix: 188f2ec)*

```
> Could not find com.android.tools.build:gradle:2.0.0-alpha1.
Searched in the following locations:
https://repo1.maven.org/maven2/com/android/tools/build/gradle/2.0.0-
   alpha1/gradle-2.0.0-alpha1.pom
https://repo1.maven.org/maven2/com/android/tools/build/gradle/2.0.0-
   alpha1/gradle-2.0.0-alpha1.jar
```

Actually, this type of build failures mostly happens on software projects that are inactive for some time. Among the 10 projects failed with removal of dependency, 7 projects have been inactive for at least 2 years, and 2 have been inactive for at least 6 months by the time we clone the code. The remaining 1 (ACRA/acra) was active, and other versions of the project do build, so we suspect the build failure may be due to a short-term mismatch between configuration file and server status, which gets fixed soon. Although the 9 projects are no longer active, they are still popular among users and users have to find the missing dependency files on the network to build these projects.

External Tools: Two projects require external tools in their build process. These external tools typically facilitate some build steps, such as creating packages in multiple formats. So, build failure happens because external tools are not in the system. Example 4 shows a project calling `fpm` command during the building process, and the `fpm` tool is not available.

BuildLogExample 4 External Tools *(gocd/gocd: a3f77f9)*

```
Execution failed for task ':installers:agentPackageDeb'.
> A problem occurred starting process 'command 'fpm''
```

3.4.2 Process Issues

Process Issues are build failures caused by the requirement of additional steps in the building process. To build these projects, we need to either run a command different from the default build command, or some additional parameters, command executions, or settings are required. 46 of the 91 build failures we found are process issues, and they fall into 2 sub-categories as follows.

Non-default Build Command: This sub-category accounts for 33 of the 46 process issues. These build failures happen because the default build command is not the correct build command required to build the project. In Example 5, we get `NoClassDefFoundError`. The project is successfully built if we enter proper the build command `mvn clean install -P 'guice'`, which activates the building profile for `guice`.

BuildLogExample 5 Non-default build command *(roboguice/roboguice: d96250c)*

```
Exception in thread "pool-1-thread-1" java.lang.NoClassDefFoundError:
   org/eclipse/aether/spi/connector/Transfer$State
at org.eclipse.aether.connector.wagon.WagonRepositoryConnector$GetTask.
   run(WagonRepositoryConnector.java:608)
```

BuildLogExample 6 Require File Setup *(google/iosched: 2531cbd)*

```
A problem was found with the configuration of task ':android:
   packageDebug'.
> File '/home/~/google_iosched/android/debug.keystore' specified for
   property 'signingConfig.storeFile' does not exist.
```

All 3 build tools we consider defines a special type of parameter that specify the type and phase of building to perform (e.g., whether just clean the project, whether perform unit testing, whether build a release or debug version of an Android apk). These parameters are called "Targets" in Ant, "Lifecycle Commands" in Maven, and "Tasks" in Gradle. In the rest of the paper, we refer to such parameters as "targets", and other parameters as "options". Since the default build command (with default target) does not work, the actual command required can be either a command with a different target or with additional options. In our study, we found 27 build failures for the former case, and 6 build failures for the latter case. The distribution of these build failures in different

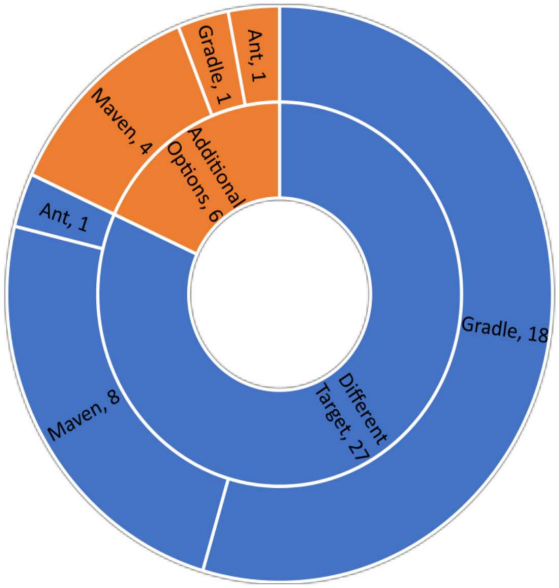


Figure 3.2: Non-Default Build Commands Distribution

build tools are presented in Figure 3.2. The figure shows that Gradle has more failures due to different targets. The reason may be that customized targets [17] (called tasks in Gradle) are used more widely in Gradle.

Require File Setup: This sub-category of 13 build failures are due to the requirement of user generated files during the build process. The two types of required files we found are local property files, in which the user should configure some options or properties such as path to SDK home, and Android keystore files, which the user should generate and sign. Example 6 shows a build failure due to the requirement of a keystore file. It should be noted that, although the user is expected to generate and sign a key with Java keytool, simply copying the default debugging key from Android SDK does not affect the correct building and execution of the project.

3.4.3 Project Issues

Project issues are build failures caused by defects in the project itself. These defects can either be code defects that prevent the software from being compiled anyway, or generalization defects that prevent the software from being built on another machine. There are 14 build failures falling into this category, and we classify them into 4 sub-categories: version conflicts in configuration

files, compilation errors, hard-coded paths, and incomplete upload.

BuildLogExample 7 Version Conflicts in Configuration Files *(google/iosched: 2531cbd)*

```
> Failed to apply plugin [id 'com.android.application']
> Gradle version 2.2 is required. Current version is 2.1. If using the
   gradle wrapper, try editing the distributionUrl in ~/gradle/wrapper/
   gradle-wrapper.properties to gradle-2.2-all.zip
```

Version Conflicts in Configuration Files: This sub-category contains 9 build failures which are caused by conflicts in the version properties in build configuration files, such as wrong gradle version defined in the wrapper file or version conflicts between parent and child `pom.xml` files in Maven. It should be noted that, since the files with correct versions already exist on the developer's machine, this particular build failure might not manifest in the developer's machine. Example 7 shows a build failure where wrong gradle version (2.1) is specified in the wrapper file. Since developers may already have gradle 2.2 installed on their machines, this defect is not observed on their machines.

Compilation Errors: The 3 build failures in this category happen due to project compilation error. The revision we fetched may be in the middle of global changes as some earlier version builds. Example 8 shows such a compilation error.

BuildLogExample 8 Compilation Failure *(daimajia/AndroidSwipeLayout: d7a5759)*

```
> Compilation failed; see the compiler error output for details.
.../library/src/main/java/com/daimajia/swipe/SwipeLayout.java:1327:
   error: illegal start of expression
  float willOpenPercent = (isCloseBeforeDragged ? ...
```

Hard-Coded Path: Hard-coded path is a common mistake preventing transplantation but it is not seen much in top projects. We do find a hard-coded path error in one project. The path is mentioned in project's configuration file and the build failure is shown in Example 9.

Incomplete Upload: For one project (Example 10), we found that one of Android resource file is missing. We suspect that the developers may forget to add the resource file into the software repository for this revision (found in other revisions).

BuildLogExample 9 Hard-Coded Path *(singwhatiwanna/dynamic-load-apk: d262449)*

```
A problem occurred configuring project ':doi-common'.
> The SDK directory '/home/~/153-singwhatiwanna_dynamic-load-apk/
   DynamicLoadApk/D:\adt-bundle-windows-x86_64-20130219\sdk' does not
   exist.
```

BuildLogExample 10 Incomplete Upload *(android/platform_frameworks_base: e011bf8)*

```
Execution failed for task ':processDebugResources'.
> com.android.ide.common.process.ProcessException: org.gradle.process.
   internal.ExecException: Process 'command '/home/~/android-sdk-linux/
   build-tools/21.1.2/aapt'' finished with non-zero exit value 1
```

3.5 Identifying Root Causes of Build Failures (RQ3)

To answer question **RQ3**, we measure for how many build failures the root causes are explicitly mentioned in the build failure logs. For build-failure identification, we used regular-expression based parser and for failure analysis we performed manual analysis on build failures. We also check whether there are build instructions in the readme files / Wiki pages of the project [97], and whether following the instructions will avoid the build failure. Specifically, we check whether a keyword (e.g. a platform version for platform-version issues) or a special object (e.g., version number) is mentioned in the build logs or readme files / Wiki pages. For example, Example 1 is considered as being revealed in the build log, while Example 14 is not.

Table 3.2 shows the proportion of build failures whose root causes can be identified from readme files, and build logs, respectively. In the table, Line 4 shows the size of the union of build failures in Line 2 and 3, and Line 5 shows the total number of build failures in the category for reference. Columns 2-10 presents the results for each category of build failures. Here we use the level 3 categories as they better reflect the property of the root cause, while level 4 categories are finer-grained and specific to build-tools or file types. Due to space limit, we use some abbreviations in the column names, but the categories in the table are in the exact same order as they are shown in Figure 3.1.

From the table, we made the following observations: First, root causes of build failures are

Table 3.2: Root Cause Revealed

Category	All	Ver. Issue	Depend. Removal	Ex. Tools	Require File Setup	Non-Default Command	Version Conflict	Comp. Error	Hard Coded Path	Incomp. Upload
Readme	12	1	0	1	0	10	0	0	0	0
Build Log	27	1	5	1	11	1	4	2	1	1
Either	39	2	5	2	11	11	4	2	1	1
All	91	19	10	2	13	33	9	3	1	1

generally difficult to find. Among 91 build failures, only 39 have their root cause or solution (e.g., the correct build command for projects using non-default commands) mentioned in the project readme files or build failure logs. Second, for several build-failure categories, it is easier to find the root cause or solutions from build log or readme files. Specifically, 11 of 13 build failures in the Require-File-Setup category have the root cause explicitly shown in the build log. The reason is that, when the required file does not exist, the build tools will always report file not found error and report the path where the required file should be placed. Also, for 10 of 33 build failures in Non-Default-Command category, the correct build command is mentioned in readme files. Although the proportion is not high, it shows the possibility of extracting the correct build command from readme files.

It should be noted that, identification of the root cause of a build failure can largely benefit the automatic software building process. However, automatic software building is still possible without knowing the root cause of the build failure. Due to the limited types and concentrated distribution of build failures as shown in Figure 3.1, it is always possible to try solutions for different root causes until build success or solutions / resources have been exhausted. For example, the automatic building tool may always try different build command targets, and recent versions of built tools / platforms to resolve build failures. Also, the vague relations between build logs and root causes, although hard to understand, may be caught by data mining techniques. Recommended root causes from build logs can save much time for automatic building tools by reducing solutions.

3.6 Automatic Resolution of Build Failures (RQ4)

To answer question **RQ4**, for each category of build failures, we study whether they can be automatically resolved with heuristics. Specifically, we performed 3 studies to show the feasibility

of automatic resolution for 3 major categories of build failures: Non-Default Command, Platform Version Issues, and Require File Setup. For the 65 build failures from these 3 categories, we are able to build 53 of them automatically which are cross validated with manual build.

3.6.1 Build Command Extraction and Prediction

To resolve the build failures caused by non-default build commands, we need to find the correct build command to execute. Table 3.2 shows that about 1/3 of projects have their correct build command in readme files / Wiki pages, which leads us to check the possibility of extracting commands directly from them.

Named Entity Recognition (NER) [55] is a well-known task to identify a specific type of entities such as people's name, location from natural language texts. It is supported by several popular NLP tool sets such as OpenNLP [52] and Stanford NLP [135]. Build commands in readme files / Wiki pages can also be viewed as a type of entities, and in a previous work [97], we proposed a technique to extract build commands from readme files and Wiki pages, and constructed a training set of readme files / Wiki pages with labeled build commands from 857 of top 1,500 GitHub Java projects which contains such build commands in readme files / Wiki pages. In our study, we apply this technique to the studied 200 projects. Note that, to avoid bias, we excluded all the studied 200 projects from the original training set.

The result of applying NER is presented in Table 3.3. The table shows that we are able to automatically build 5 projects whose correct build command is in readme files / Wiki pages, which is half of the projects having correct build command available. Example 11 shows an example of resolved build failure and the readme files containing the correct build command. Our NER based command extraction tool can extract proper command "mvn clean install" and build the project successfully.

For the projects whose correct build commands are not in the readme file / Wiki pages, we found that most of them use a non-default target but do not need additional options. For Ant, Maven and Gradle if we executed command "ant", "mvn" and "gradle –tasks" respectively, we can

obtain the list of all available build targets in the project. But we still need to choose the correct target to use if we do not want to try them exhaustively. Based on our large training set from 857 projects with manually labeled build commands, we are able to find which target is more like the correct building target by calculating the similarity between the target name and all the extracted commands in our training set under the same build tool. Note that this technique is very simple and just taking into consideration the target name, but with the top target we fetch, we are able to successfully build 21 of 24 projects that fails due to using non-default target and not solved with NER. The detailed result is also shown in Table 3.3.

BuildLogExample 11 Failure resolved with NER *(apache/storm: 3a5ecf5)*

```
Readme File:
The following commands must be run from the top-level directory.
mvn clean install
If you wish to skip the unit tests you can do this by adding -
   DskipTests to the command line.
================================================
Build Log:
[ERROR] Failed to parse plugin descriptor for org.apache.storm:storm-
   maven-plugins:2.0.0-SNAPSHOT (~/storm-buildtools/storm-maven-plugins
   /target/classes): No plugin descriptor found at META-INF/maven/
   plugin.xml -> [Help 1]
```

Example 12 shows an example of build failure resolved with target estimation. In the list of available non-default targets, we selected "assembleDebug" which has highest ranking. It builds the project successfully because it does not look for API Key, which is required by "build" target and fails the build.

BuildLogExample 12 Failure resolved with Estimation *(HannahMitt/HomeMirror: 71c860)*

```
ERROR - Crashlytics Developer Tools error.
java.lang.IllegalArgumentException: Crashlytics found an invalid API
   key: null.
Check the Crashlytics plugin to make sure that the application has been
    added successfully
Contact support@fabric.io for assistance.
at com.crashlytics.tools.android.DeveloperTools.processApiKey(
   DeveloperTools.java:375)
```

3.6.2 Version Reverting

Executing build command with parameter estimation in many cases failed due to incompatible SDK and build tools(e.g Maven, Gradle) versions. To handle such issues, a straightforward way is to revert the versions of SDK and build tools. To study the cost of doing so, we implemented a tool to automatically perform the version reverting, and try to find out how many versions we need to try before finding the correct version. Table 3.4 presents the result of this study, in which rows 2-5 shows the number of projects whose build failures are resolved within a certain type of version reverting, and rows 6-7 shows the average and maximal reverted versions for resolving build failures. The result shows that all 19 failures can be resolved within 10 version reverting, and the average reverting required is 3.8.

However, since it is not always easy to find out which build tool or SDK has version issues, so the automatic building tool may need to try them one by one, and a sum of Java, Maven, and Android trials will bring the worst case to 15 trials.

3.6.3 Dummy File Generation

For build failures in the category of Require-Data-Setup, it is easy to find their root cause (typically shown in the build failure log). Therefore, we can always try to generate a dummy local file as a place holder. Also, in many projects, a sample local file (e.g., local.property.example) is provided, and users can refer to it for what to be put into the local file. In our study, we find that, simply generating an empty local file will resolve 7 of the 13 build failures, and renaming the sample local file (the file whose name is closest to the required file) back will resolve 1 additional build failures.

3.6.4 Other Types of Failures

The other types of failures may not have general and straightforward resolution. Removed Dependency and Config Version Conflict are two other large sub-categories with 19 build failures in total. Removed Dependency failures can be easily resolved if the dependency file can be found

Table 3.3: Resolved Build Failures with Command Extraction and Prediction

Build Tool / Sub-Type	Maven	Gradle	Ant	All	Target	Para
NER	2	3	0	5	4	1
Target Estimation	5	15	1	21	21	0
All fixed	7	18	1	26	25	1
Not fixed	5	1	1	7	3	4

Table 3.4: Resolved Build Failures with Version Reverting

Platform	Java	Maven	Gradle	Android	All
Revert 1 version	9	0	0	0	9
Revert 2-5 versions	0	0	1	2	3
Revert 6-10 versions	0	6	1	0	7
Revert 11+ versions	0	0	0	0	0
Max # reverted versions	1	10	9	4	10
Avg # reverted versions	1	7.3	6.5	3	3.8

in large Jar repositories (e.g., Maven Central, Java2S) or Google, but the difficulty varies for each Jar. A potential solution is to search for references to the Jar file in other projects' configuration files, and try to fetch the Jar file from their project folder or referred server.

A large portion of Config Version Conflict failures (7 of 9) are due to out-of-date Gradle wrappers. These failures are easy to find and resolve, because we just need to update the version specified in gradle wrapper to the same as the gradle script. However, the other 2 failures are due to mismatches between parent and child pom files in maven. To resolve them, we need to perform in-depth analysis of pom files and their dependencies.

3.7 Discussions

3.7.1 Threats to Validity

The major threat to the internal validity of our evaluation is the correctness of manual process in our experiment, including the implementation of our approach, the labeling of build commands, and the manual build process. To reduce this threat, we carefully performed all these process, and double checked the consistency throughout the data sets. The major threat to the external validity of our evaluation is that our findings may be only applied to our subject data set. To reduce this threat, we used a large number of top Java projects for evaluating project building. Our experiment

confirms that these projects cover various build configuration systems.

3.7.2 Lessons Learned for Automatic Software Building

It is a necessity. Our study finds that, about half of the top Java projects cannot be straightforwardly built with default build commands. This is a very disconcerting fact: consider having to create a data set of 200 built Java projects (not very large for mining or training purposes), while a large number can be automatically built with simple default build commands, a researcher still needs to manually build 100 projects.

Furthermore, referring to our experiment on top 200 projects again, among the 86 projects that have build failures, 59 do not contain sufficient documentations (e.g., in readme files) that describe the correct building instructions or common building pitfalls. A developer wanting to build such projects have little choice but to engage in a "trial by error" attempt to manually iterate through known build targets. This is obviously a time consuming process and in no way a scalable process, if done manually. It is a clear challenge that has to be addressed if software engineering researchers aspire to develop large-scale program analysis techniques that require massively large sets of build artifacts. Perhaps one can consider simply to rely on a best effort automated use of default build commands to build half of the projects in a large corpus. This naive solution however, might introduce unknown bias into downstream research processes. In essence, this 'build' problem is not simply just a matter of productivity, but likely a hurdle of the software engineering research methodologies.

It is feasible. Our study shows that, among the 86 projects with build failures, 26 projects can be built successfully with relatively simple heuristics. For instance, reverting platform to all possible versions, and generating a dummy property file. Furthermore, 26 additional projects can be built successfully by the NER technique and mining commands from a large set of readme files. The 7 projects with unmatched gradle wrappers and missing dependency Jar files also have the potential to be automatically built by fixing the mismatches between gradle wrapper and gradle scripts, and searching and downloading the missing Jar files from other servers (e.g., Java2S [21]). These

numbers show that, more than 75% of the projects with build failures can be automatically resolved with simple rules or some advanced techniques, so a tool combining default command execution and the resolution techniques above may achieve an overall build successful rate of 80%. A tool that systematically binds together these heuristics to fix build scripts and strategies to iteratively try build alternatives, would not only reduce the manually effort of software engineering researchers, as well as enlarge the amount of build data that they can extract from open repositories.

The challenges. Our study has also identified several build-failure categories whose automatic resolution can be difficult. For instance, when the building process requires external tools, when a dummy local property file cannot enable successful build, or when complicated build commands are required but no instructions are available in the readme files / Wiki pages. To handle these cases, a tool needs to automatically analyze the project files structures, build configuration files, and also mine and analyze discussion threads from online forums such as StackOverflow [32]. These research tasks not only can be challenging but also of great value. Note that bringing the overall build successful rate from 80% to 90% will reduce half of the manual build effort required by researchers.

Another challenge for automatic building tools is the time and resource consumption of building each project. A successful build is possibly preceded by a large number of unsuccessful attempts. For example, the building tool may try a lot of build-tool versions to find the version that is compatible with the project. Although the build process is automated, considering the large number of projects in open repositories, the performance can still be a major concern. Hence, it is very important that the techniques that we use must be easily scalable (i.e., large volumes of input can be handled by adding more compute strength to the cluster). Also these techniques must be highly optimized, for instance, in the manner they extract/use information from build failure logs and possibly learn from previous attempts (e.g., recognize and prune away known hopeless attempts).

3.7.3 Lessons Learned for Build-Tool Developers

Our study has shown that the backward incompatibility of build tools is one of the leading root causes of build failures. As time goes by, a project will no longer be buildable with the newest version of built tools. While backward incompatibility is often unavoidable, build tool developers should maintain and make available all previous versions of the tool, and force the build configuration file to specify the version that the project is built on. In fact, Gradle Wrapper [18] is a built-in mechanism in Gradle that automatically downloads and uses the proper gradle version. Interestingly, we found out-of-date gradle wrapper files in 7 projects, and the presence of such fault in a project often fatally derails the building process. We believe that these consistencies between the wrapper and the build script should be enforced by the build tool itself. Furthermore, a lot of projects are using non-default build commands and parameters without having any instructions in their readme files. An excellent feature that build tool developers should consider supporting is one that records the developers' command sequences when they build the software and automatically generate some wrapping scripts that repeats their build actions.

3.7.4 Lessons Learned for Project Developers

The Version Conflicts in Configuration Files build failure category contains 9 build failures that are caused by defects in the software project itself. The 3 code compilation errors are due to incomplete commits (the code revision we downloaded is in the middle of a global change and thus cannot be built). The rest build failures are all caused by defects that prevent the software to be built on another machine, such as hard-coded paths, miss-uploaded files, and out-of-date version in wrapper files or configuration files. This calls for a testing of the project on a different machine when a change is made to configuration files / wrapper files or build dependencies. In fact, this can be forced by some code reviewing tools like Gerrit [11]. And the automatic software building tools can also serve as a testing tool to report build-related defects (such as out-of-date gradle wrappers) to project developers.

3.7.5 YML Files and Continuous Integration

For Continuous Integration (CI) [102], some projects adopt Travis CI and include a yml file in the project repository to specify build steps for CI process. In our study, we find 95 of the top 200 projects to include such yml files. Using yml files to build the project requires specific integration configurations and software support from Travis CI, so in our study we do not use yml files for automatic building.

3.8 Related Works

Study on Build Failures. On the study of building errors, Hyunmin et al. [178] carried out an empirical study to categorize build errors at Google. Their study shows that missing types and incompatibility are the most common type of build errors, which are consistent with our findings. However, they also find many semantic or syntactic errors, which are very rare in our study. This is not surprising, since their study focused on the build errors in the original environments, while our study focuses on the build errors due to environment changes. Also, since our projects are committed versions in the repository, with support of current IDEs, they are more likely to be built successfully in original environment, and have fewer code related errors. Recently, Tufano et al. [195] studied the frequency of broken (not compilable) snapshots and likely causes of broken snapshots. Sulír et al. [187] performed build failure analysis based on build log text categorization to find out frequency and reasons for build errors. McIntosh et al. [142]'s study also supports that for modern build systems build maintenance effort on external dependency is higher than than internal dependency management. Compared to these works which analyze only build logs, we performed detailed manual analysis and building to find out and confirm the root causes of the build failures. As an example, dependency issues are found to be a major reason of build failures by previous works, but we found that such dependency issues may be caused by build-plug-in version errors or a wrong build command used. So simply adding the dependency back to the project may not resolve the build failure.

Automatic Software Building. Lämmel et al. performed semi-automatic building of Ant-based

Java projects (i.e., the QUAATLAS corpus [76]) in their API statistics study [117] . The thesis version [185] of the work details the software building process, which includes automatic scripts to locate Ant configuration files and to run the Ant command. However, the work uses only pre-defined Ant commands sets, and does not take advantage of information in readme files, so it is similar to our baseline approach but specific for Ant. On migration of build configuration files, AutoConf [12] is a GNU software that automatically generates configuration scripts based on detected features of a computer system. AutoConf detects existing features (e.g., libraries, software installed) in a build environment, and configure the software based on pre-defined options. Most recently, Gligoric et al. [88] proposed an approach to automate the migration of various building configuration files to CloudMake configuration files based on building execution with system-level instrumentation.

Analysis of Building Configuration Files. Analysis of build configuration file is growing as an important aspect for software engineering research such as dependency analysis for path expression, migration of build configuration file and empirical studies. On dependency analysis, Aoumeur [51] proposed a Petri-net based model to describe the dependencies in build configuration files. Adams et al. [46] proposed a framework to extract a dependency graph for build configuration files, and provide automatic tools to keep consistency during revision. Most recently, Al-Kofahi et al. [48] proposed a fault localization approach for make files, and SYMake [188] uses a symbolic-evaluation-based technique to generate a string dependency graph for the string variables/constants in a Makefile , automatically traces these values in maintenance tasks (e.g., renaming), and detect common errors.

3.9 Conclusions

Software building takes significant amount of time for software engineering researchers before analyzing projects. Developers may also need to build a list of third party tools before they can use them. This paper comes up with the first study on the build failures found in top Java projects, and whether such failures can be resolved with automatic tools. We have constructed a taxonomy

for the root causes of build failures, and study the distribution of build failures in different categories. Specifically, we found 91 build failures in 86 of the 187 Java projects that use Maven, Ant, and Gradle for their building process, and we found the leading root causes of build failures are backward-incompatibility of JDK and building tools, non-default parameters in build commands, and project defects in code / configuration files. Finally, 52 of the build failures can be resolved automatically, and additional 6 build failures have the potential to be resolved automatically.

Acknowledgments This material is based on research sponsored by NSF Award CCF-1464425 and DARPA grant under agreement number FA8750-14-2-0263.

CHAPTER 4: CHANGE-AWARE BUILD PREDICTION MODEL FOR STALL AVOIDANCE IN CONTINUOUS INTEGRATION

In this chapter, we discussed the build prediction model that uses TravisTorrent data set with build error log clustering and AST level code change modification data to predict whether a build will be successful or not without attempting actual build so that developers can get early build outcome result. This work has been presented at the following venue [96]:

- F. Hassan and X. Wang, "Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, 2017, pp. 157-162.

4.1 Introduction

Software development process and speed have changed drastically over the years with more distributed development teams. With the requirement growth, software development teams need to adopt novel practices with evolving social coding and process automation platforms. These practices allow distributed teams to work closely and help to increase productivity. Projects that are more popular, are subject to higher delivery pressure and more frequent release cycle, and thus require more rapid development, testing, integration etc. Broken integration will slow down the release process, and as a result we need to have more centralized integration process.

To ensure the system is functioning with every code changes made by the developers, one key innovative idea is Continuous Integration(CI) [198] and this process has been adopted by many organizations for faster integration. A typical CI system attempts to automate the whole software build process from compilation to test case execution. For CI systems, dedicated infrastructure with different build systems such as Make, Ant, Maven, and Gradle are used to automate the building process. Despite the growing interest in CI, build failures in respect to CI process is an under-explored area. Even though CI is used for continuous development, the integration process might be delayed for long build chains, build error fixes, and frequent code changes in version

control system. According to analysis on TravisTorrent [62] data, the median build time for Java project is over 900 seconds and the median length of continuous build failure sequences is 4. Thus when multiple developers are committing their changes concurrently, their commits may be piled up in the building queue, and they may need to wait for a long time to get the build feedback. They also have the option to continue working on their copy, but they will be at the risk of rolling back their changes if their original commit fails the integration.

Therefore, it is desirable to have a recommendation system that predicts the build feedback of a code commit and thus gives developers more confidence to continue their work and reduce the chance of rolling back. In this paper, we analyzed software build execution time, commit and consecutive build status change to study the necessity and possibility of a change-aware build prediction model. Furthermore, we propose a recommendation system to predict build outcome based on the TravisTorrent data and also the code change information in the code commit such as import statement changes, method signature changes to train the build prediction model.

To evaluate our approach, we conducted an experiment with TravisTorrent data set. We focused on Java projects using Ant, Maven and Gradle build systems because they are supported by the TravisTorrent data set. Our evaluation results show that our model can can achieve an average F-Measure of 87% on all three build system for cross-project build-outcome prediction, which is a very challenging but more realistic usage scenario.

Our paper makes the following main contributions.

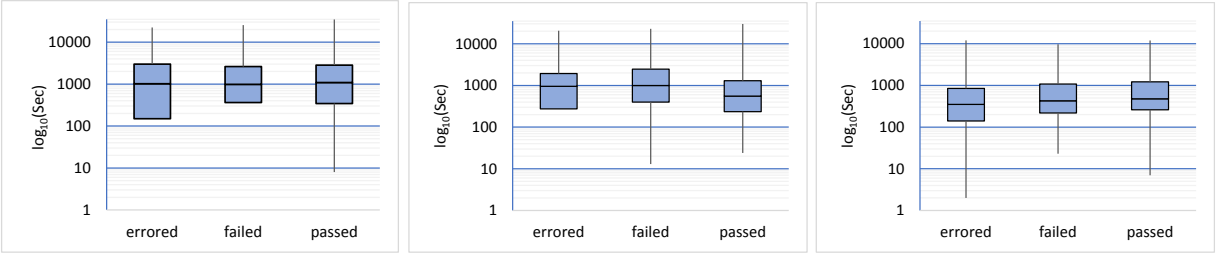
- A statistic study of CI build status and time on the TravisTorrent data set to motivate our work.
- A build-outcome prediction model based on combined features of the build-instance meta data and code difference information of the commit.
- A large-scale evaluation of our project with both scenarios of cross-validation and cross-project prediction on the TravisTorrent data set with more than 250,000 build instances.

The rest of this paper is organized as follows. We first introduce some existing research efforts

in Section 6.8. Then, we present an empirical study on the building status of the data set in Section 4.3. Our prediction model is presented in Section 4.4, followed by our evaluation in Section 4.5. Before we conclude in Section 6.9, we discuss the threats to the validity of our evaluation in Section 4.6.

4.2 Related Work

Over decades, researchers have been worked on defect prediction models [230] to guide software testing [148, 202] and quality assurance [184, 190, 203]. These defect prediction models are used to identify early defect prone components and thus reduce development time and also reduce maintenance cost. These models are general and does not consider features specific for software builds. Build co-change prediction models [219] are used to identify when we need to change build configuration files to avoid possible build failures, but does not predict build outcome. Wolf et al. [215] and Irwin et al. [114] utilized social network analysis and Socio-technical analysis to predict possible build failures in a project. Bird and Zimmerman [63] discussed software build error prediction and potential approaches, but they did not provide detailed techniques and evaluation. Finalay et al. [85] proposed an software-metrics-based approach to predict build output, but it requires design and history features so that cannot be easily applied to unprepared projects. These works considers isolated workstation environment for prediction model, but CI environment has different work-flow and can suffer with different latency as discussed in different studies [102] [61]. Recently, we [94] studied the reasons of build failures in a large number of projects. Ansong and Ming proposed build outcome prediction model [153] for predicting build fail in CI environment based on commit information of current and last push with statistical information. Our approach uses build commit information, build error type and code change metric to predict build outcome prediction in CI environment with better performance than previous work [153].



(a) Ant Build Execution Time (b) Maven Build Execution Time (c) Gradle Build Execution Time

Figure 4.1: Ant, Maven and Gradle Build Execution Time Statistics

4.3 DATASET AND CHARACTERISTICS

In this paper we studied TravisTorrent data set [62] on Oct 27,2016. This data set includes 402 Java projects with data for 256,055 build instances. Among this 256,055 build instances, the Ant based build system is used for 104,417 cases, while Maven and Gradle based build systems are used for 104,876 and 44,056 instances, respectively. For better analysis and build prediction model, we also used raw build logs of corresponding data set from TravisTorrent ftp server.

4.3.1 How Much Time Required for Building?

In the TravisTorrent data set, there are four types of build status: passed, failed, errored and canceled. Canceled build status denotes that build process is interrupted while build in progress. So, we simply ignore this status in our study. Then, we considered passed build status as a successful build, while failed and errored build status both are considered as failed build. For each category of build status, we considered three most popular build systems: Ant, Maven and Gradle and performed analysis on build execution time for Ant, Maven and Gradle build system with different build status. As shown in Figure 1, the median build execution time for errored status with Ant, Maven and Gradle build tool are 1,019, 955 and 352 seconds respectively. For failed build status, the median execution time for Ant, Maven and Gradle are 981, 998, and 426 seconds, respectively. Passed build execution time for Ant, Maven and Gradle are 1,090, 558 and 477 second respectively. The maximum build execution time for each build status for each build tool are much higher than median execution time. For example, Ant and Maven tool with errored and failed status takes

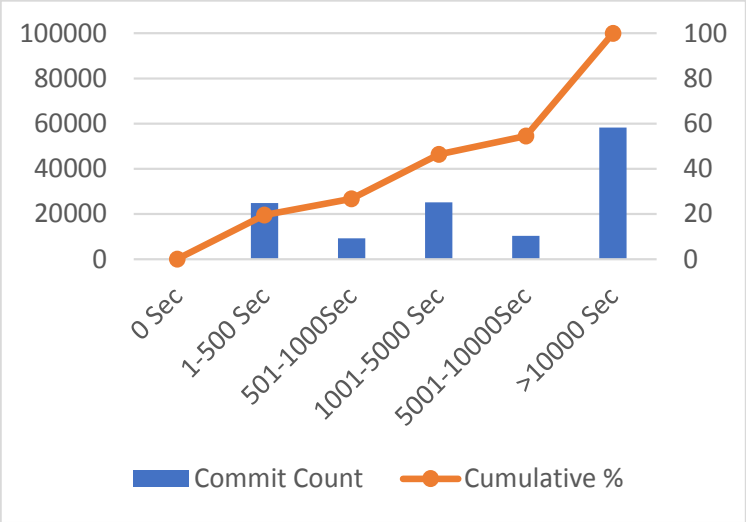


Figure 4.2: Commit Frequency Time Distribution

over 20,000 seconds, while for Gradle build tool it takes over 9,000 seconds for errored and failed build status. So, developers typically need to wait for a long time to get the build feedback and it creates stall in the CI process which is also supported by other studies [226].

4.3.2 What is the Time Interval in Between Two Commit?

Code commit time interval between two consecutive commits within the same project can tell how often developers need CI feedback. We performed commit interval frequency based on time interval of two subsequent commits of TravisTorrent data set. According to our analysis shown in Figure 4.2, 19.54% code commits occurs within 500 seconds, 26.74% of commits occurs within 1,000 seconds, and 54.53% of commit occurs within 10,000 seconds. According to Section 4.3.1, the median build execution time is 500 to 1,000 seconds. So, for about 20% of the code commits, software builds will line up in the building queue and cannot be performed immediately.

4.3.3 How Often Consecutive Build Status Changes?

We performed analysis on how often build status changes, or for how many consecutive code commits, the build status remains unchanged. Figure 4.3 shows the statistical result on consecutive build status for errored, failed and passed. For errored and failed build status, build outcome

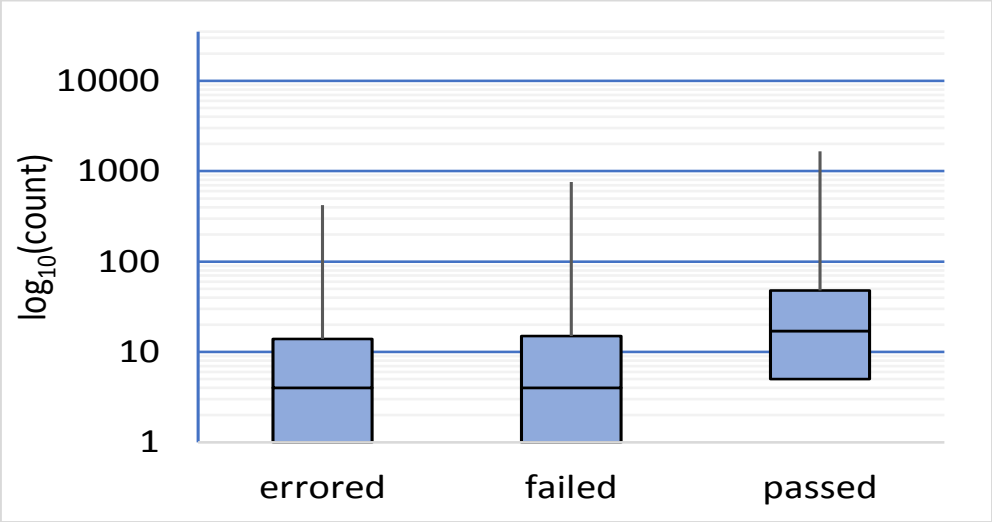


Figure 4.3: Build Status Statistics on Consecutive Build Sequence

remains unchanged with median of four build instances. For maximum case, 422 consecutive build was errored, while 760 consecutive build was failed. Such consecutive errors and failures imply that developers keep working on their parts in spite of the build failures, or without knowing about the build failures.

4.4 Overview of Build Prediction Model

The overall architecture of build prediction model is shown in Figure 6.2. Our approach consists of three parts: data filtering, feature generation, and model generation. In the data filtering phase, we extract useful information from the build logs. Then we generate features from the extracted data and the code change information in the code commit to be predicted. With the features, we train and evaluate our predict model with Weka [91].

4.4.1 Data Filtering

For our research, we considered Java projects that uses Ant, Maven and Gradle to predict build outcome prediction model. So, filtered Java projects that uses Ant, Maven and Gradle for our research consideration. Based on the filtered build data, we performed feature generation and model generation to predict build outcome in CI environment.

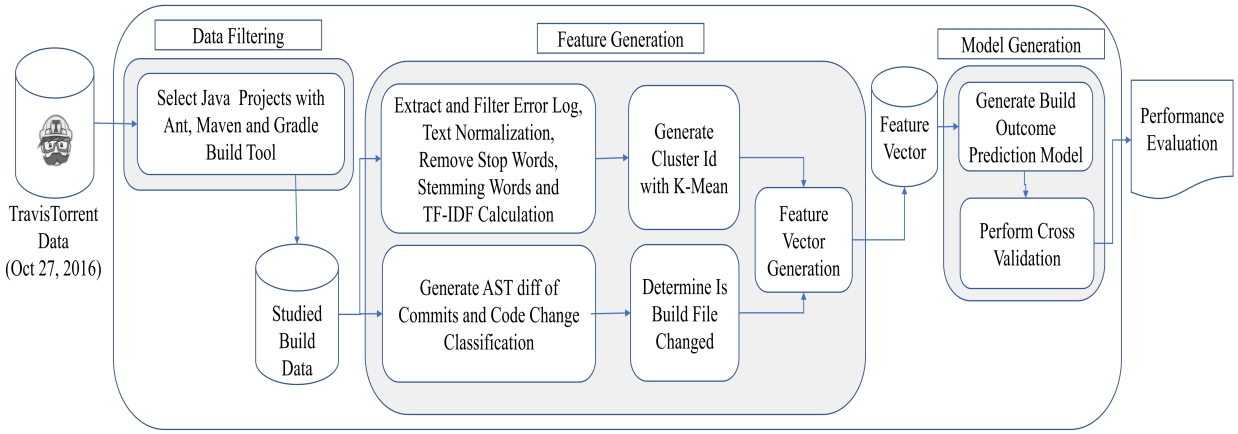


Figure 4.4: Overview of Build Prediction Model

4.4.2 Feature Generation

Build Failure Cluster ID Generation

Although successful builds are almost always the same, build failure can happen for different reasons such as compilation issues, dependency issues, or tools version issues. Information about different types of build failures are useful for predicting the successfulness of the following builds. In our approach, we fetched raw build logs from TravisTorrent ftp server for errored and failed build status and performed document clustering to group similar build failures in same group and uses cluster id to denote each failure group. During cluster id generation, we first filter build log text of Ant, Maven and Gradle build logs. Build logs denote different activities such as downloading dependencies, compiling source files, errors if happening during build. We are mainly interested in errors and exceptions in build log file, because they are more relevant to the type of build errors. For passed build instances, we considered them as a single cluster as passed.

Specifically, we developed regular expression based parser for Ant, Maven and Gradle to extract only error and exception log parts. After filtering, we performed text normalization and stop word removal process. Then we performed stemming with popular Porter stemming algorithm [214]. After that, we calculated Term Frequency–Inverse Document Frequency (TF–IDF) [170] for denoting statistical uses of a word in given document(s) to reflect the importance of the word.

With TF-IDF value, we performed clustering using popular K–Mean Clustering [186] algorithm. For K–Mean clustering value of K is calculated using formula $\sqrt{n/2}$, where n is the number of build logs. We calculate the cosine similarity of the query's vector and the cluster centroid's vector. Among the generated clusters compilation failures, test failures, and dependency resolve failures are most prominent clusters. Examples 1 and 2 show build log part of two different project having dependency resolve failure issue are clustered in same group with our approach and denoted by same cluster id.

BuildLogExample 13 *(gh_project_name: BuildCraft/BuildCraft, tr_build_number: 954)*

```
A problem occurred configuring root project 'BuildCraft'.
> Could not resolve all dependencies for configuration ':classpath'.
```

BuildLogExample 14 *(gh_project_name: MrTJP/ProjectRed, tr_build_number: 453)*

```
Could not resolve all dependencies for configuration ':compile'.
> Could not resolve codechicken:CodeChickenLib:1.7.10-1.1.1.104.
```

Features from Code Commits

To further consider features from the source code change of the code commit, for each build instance, we extracted the commit hash from TravisTorrent data set. For each commit, we further explore the changes with JGit [22], a Git client for Java, and generated the AST diff of the code commit with GumTreeDiff [83]. Based on the the AST diff of the code commits, we build features: `Import Statement`, `Class Signature`, `Attributes`, `Method Signature` and `Method Body` changes and count the number of changes as feature value. Apart from above Java code change features, we also considered file structure level features, such as how many build script such as pom.xml, build.gradle and build.xml files are changed.

4.4.3 Model Generation

Feature Selection for Model Generation

We applied Information Gain(IG) attribute evaluation [154] algorithm to select discriminating features from the feature set. For information gain attribute, entropy is a commonly used for characterizing the purity of information. Entropy is is used for IG attribute ranking methods. The entropy measure is considered as a measure of system's unpredictability. The entropy of Y is

$$H(Y) = -\sum_{y \in Y} p(y) log_2(p(y)) \tag{4.1}$$

where p(y) is the probability density function for the random variable Y. Information Gain Attribute Evaluation entropy value resides in between 0 to 1. Higher entropy value indicates higher effectiveness of the feature. During feature selection we considered four features of the previous build instance, and all remaining features are from the current build instance. Table 4.1 shows the list of used features. Besides the existing TravisTorrent features (marked as TravisTorrent in Column 3), we also generated other features marked as Generated. Description of our generated features are provided in Table 4.2.

Build Prediction Classifier Construction

We construct classifier using the random forest algorithm [65] of Weka implementation. The random forest classifier produces distinct decision trees which are random subset of all model attributes, and we use random forest algorithm because the algorithm has been reported to be most effective in existing software engineering research efforts [219]. The classifier calculates a classification decision for each of the trees and then aggregates the partial results to a total classification result.

Table 4.1: Features Used for Build Prediction Model

Build Instance	Feature Name	Source of Feature
Previous Build Instance	prev_bl_cluster	Generated
	prev_tr_status	TravisTorrent
	prev_gh_src_churn	TravisTorrent
	prev_gh_test_churn	TravisTorrent
Current Build Instance	gh_team_size	TravisTorrent
	cmt_buildfilechangecount	Generated
	gh_other_files	TravisTorrent
	gh_src_churn	TravisTorrent
	gh_src_files	TravisTorrent
	gh_files_modified	TravisTorrent
	gh_files_deleted	TravisTorrent
	gh_doc_files	TravisTorrent
	cmt_methodbodychangecount	Generated
	cmt_methodchangecount	Generated
	cmt_importchangecount	Generated
	cmt_fieldchangecount	Generated
	day_of_week	Generated
	cmt_classchangecount	Generated
	gh_files_added	TravisTorrent
	gh_test_churn	TravisTorrent

Table 4.2: Generated Feature Description

Feature Name	Description of Feature
prev_bl_cluster	Previous Build Cluster ID
cmt_buildfilechangecount	Number of build script file change
cmt_methodbodychangecount	Number of method body change count
cmt_methodchangecount	Number of method signature change
cmt_importchangecount	Number of import statement changes
cmt_fieldchangecount	Number of class attribute change
day_of_week	Day of week of the first commit for the build
cmt_classchangecount	Number of class changed

Table 4.3: Performance Evaluation of Build Prediction Model

Build Tool	Precision	Recall	F-Measure	ROC Area	Class/Type
Ant	0.948	0.948	0.948	0.975	Pass
	0.923	0.924	0.923	0.975	Fail
	0.938	0.938	0.938	0.975	Weighted Avg.
Maven	0.960	0.963	0.961	0.950	Pass
	0.838	0.825	0.831	0.950	Fail
	0.937	0.937	0.937	0.950	Weighted Avg.
Gradle	0.945	0.956	0.951	0.936	Pass
	0.830	0.794	0.812	0.936	Fail
	0.921	0.922	0.921	0.936	Weighted Avg.

4.5 Evaluation and Result Study

In this section, we evaluated our proposed model on Java projects that uses Ant, Maven or Gradle from TravisTorrent data set [1]. During evaluation we tried to answer following research questions.

- **RQ1:** To what extent our build prediction model can successfully predict build outcome for Cross Validation and Cross Project prediction model?

- **RQ2:** Which feature attributes of our models are important for build outcome prediction in CI environment?

To address **RQ1**, we evaluated our build prediction model for Ant, Maven and Gradle build systems. As build errors of Ant, Maven and Gradle are different and build-log formats are different for these tools, we construct separate model for Ant, Maven and Gradle. We performed 8-Fold Cross Validation with feature set mentioned at Table 4.1 with random forest learning algorithm for Ant, Maven and Gradle build prediction. Table 4.3 shows build outcome prediction model performance and Table 4.4 shows confusion matrix of prediction model to give better idea of what our classification model is getting right for pass and fail build status.

[1]The traing, testing and SQL dump files used in our evaluation are available at https://drive.google.com/drive/folders/0B8nZRb4sCPS_RUF4SWtFa0tIQUU?usp=sharing

Table 4.4: Confusion Matrix of Build Prediction Model

Build Tool	Actual: Pass	Actual: Fail	
Ant	51766	2864	Predicted: Pass
	2826	34121	Predicted: Fail
Maven	70071	2692	Predicted: Pass
	2952	13907	Predicted: Fail
Gradle	29330	1338	Predicted: Pass
	1694	6535	Predicted: Fail

Table 4.5: Cross Project Performance Evaluation of Build Prediction Model

Build Tool	Precision	Recall	F-Measure	ROC Area	Class/Type
Ant	0.920	0.918	0.919	0.938	Pass
	0.907	0.909	0.908	0.938	Fail
	0.914	0.914	0.914	0.938	Weighted Avg.
Maven	0.929	0.949	0.939	0.927	Pass
	0.853	0.802	0.827	0.927	Fail
	0.909	0.910	0.909	0.927	Weighted Avg.
Gradle	0.908	0.919	0.913	0.873	Pass
	0.777	0.752	0.764	0.873	Fail
	0.872	0.873	0.872	0.873	Weighted Avg.

According to Table 4.3 for both Ant and Maven average Precision, Recall, F-Measure and ROC Area are above 0.93. While for Gradle, average precision, recall and F-Measure is above 0.92. Confusion Matrix provided at Table 4.4 gives idea about successful prediction rate of our model for both pass and fail classes. According to the table, Ant build tool successfulness for pass class is 94% and fail class is 92%. For Maven successfulness for pass class is 95% and fail class is 83%, while for Gradle successfulness for pass class is 94% and fail class is 83%. For all build tool, fail class instances are less than pass class instances in training and testing. As result, performance for fail class prediction rate is less than the pass class prediction.

Apart from 8-Fold Cross validation, we also performed Cross Project validation, in which for each build system (Ant, Maven, and Gradle) 80 percent of the projects (alphabetically higher ranked) are used as training sets and the remaining 20 percent of the projects are used as testing sets. Table 4.5 shows performance evaluation of cross-project validation and Table 4.6 shows confusion matrix of cross-project evaluation.

Table 4.6: Confusion Matrix of Cross Project Build Prediction Model

Build Tool	Actual: Pass	Actual: Fail	
Ant	4186	376	Predicted: Pass
	366	3668	Predicted: Fail
Maven	10658	569	Predicted: Pass
	811	3290	Predicted: Fail
Gradle	3069	272	Predicted: Pass
	312	946	Predicted: Fail

Table 4.7: InfoGainAttributeEval Entropy for Ant, Maven, Gradle and Average for Top Ten Features

Feature Name	Ant	Maven	Gradle	Avg
prev_bl_cluster	0.6661	0.3963	0.3811	0.4812
prev_tr_status	0.6444	0.3893	0.3750	0.4696
gh_team_size	0.0403	0.0181	0.0354	0.0313
gh_src_churn	0.0141	0.0050	0.0044	0.0078
prev_gh_src_churn	0.0089	0.0045	0.0043	0.0059
cmt_buildfilechangecount	0.0049	0.0044	0.0083	0.0058
cmt_importchangecount	0.0105	0.0060	0.0007	0.0057
cmt_methodbodychangecount	0.0113	0.0039	0.0016	0.0056
gh_test_churn	0.0083	0.0079	0.0004	0.0056
prev_gh_test_churn	0.0084	0.0074	0.0006	0.0055

For Cross Project evaluation, the effectiveness of build prediction models drop a bit, but for Ant and Maven it can still predict build outcome with over 0.90 F-Measure. For Gradle, our build prediction model can predict build outcome with over 0.87 F-Measure. Successfulness for pass and fail class for Ant build is 91% and 90% respectively. For Maven, correctly predicted pass class is 92% and fail class is 85%. While for Gradle pass class successful perdition rate is 90% and fail class successful prediction rate is 77%.

To answer **RQ2**,during feature selection we applied Information Gain Attribute Evaluation on Ant, Maven and Gradle data set and select those attributes having average entropy>0.005. Table 4.7 shows the top ten features features among the used features for our model generation with average entropy from high to low.

Among the feature set, prev_bl_cluster and prev_tr_status are most prominent. prev_bl_cluster feature denotes build failure category type. For example, build failure category type can be de-

pendency missing and that case build it might requires to change source code import statement update or build configuration file update. So, prev_bl_cluster feature gives high entropy value to other feature such cmt_importchangecount,cmt_importchangecount etc. and considered as most prominent feature. Apart from that, prev_tr_status feature is also considered as important feature. In most cases if previous build status is passed, then next build has higher probability of passing. Similarly, build failure has long chain of continuous build failure.

4.6 Threats To Validity

Regarding the validity of our experiment, we identify the following threats to the internal, external and construct validity.

Internal Validity. One threat to internal validity is related to training and test set selection. We tried to mitigate the issue with cross-validation and cross-project validation. Even after that there might be issue pass and fail class imbalance due to nature of build sequence pattern. Others threat to internal validity might be during feature selection. During feature selection, we tried to select feature from TravisTorrent data set and also we generated other features related to code change and build error type. There might have other attributes that could be helpful for build outcome prediction.

External Validity. Our experimental results might have concerns of generalizability, since we performed the experiments with TravisTorrent projects those adopted TravisCI for continuous integration. However, further experiments with commercial systems, projects with different programming languages and systems using other build tools instead of Ant, Maven and Gradle can give generalized result.

Construct Validity. To generate prediction model, we grouped similar failure cases in same group using clustering algorithm. Due to different text pattern of build logs, our approach might fail to group similar bugs in same group. We tried to mitigate the issue using regular expression based filtering of build log text. Apart from that during code change classification we used GumTree. GumTree has been widely used for AST level code diff. But due to versatile pattern of code

change, there can have issues that we might failed to extract appropriate code change type.

4.7 Conclusion and Future Work

Although several approaches have been developed for predicting build co-changes or component failure, we propose the scalable approach for predicting build outcome in CI environment with evaluation on large scale data. Our evaluation shows that our approach predicts build outcome with over 87 percent F-Measure for all build systems in CI environment. Our approach will help developers to get early build outcome without making actual build. This model even can also be helpful for reducing CI computation resources. For current build prediction model we considered code change of source file, not build configuration file. In future, we are planning to use build configuration change type as feature for build outcome prediction model. Apart from that, different learning algorithms can be used for better accuracy.

Acknowledgments The authors are supported in part by NSF Grant CCF-1464425.

CHAPTER 5: HIREBUILD: AN AUTOMATIC APPROACH TO HISTORY-DRIVEN REPAIR OF BUILD SCRIPTS

In this chapter, we discussed HireBuild: History-Driven Repair of Build Scripts, the first approach to automatic patch generation for build scripts, using fix patterns automatically generated from existing build script fixes and recommending fix patterns based on build log similarity. A significant portion of this work has been presented at the following venue [99]:

- F. Hassan and X. Wang, "HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, 2018, pp. 1078-1089.

5.1 Introduction

Most well maintained software projects use build tools, such as Ant [192], Maven [145] and Gradle [105] to automate the software building and testing process. Using these tools, developers can describe the build process of their projects with build scripts such as `build.xml` for Ant, `pom.xml` for Maven, and `build.gradle` for Gradle. With growing software size and functionality, build scripts can be complicated [141] and may need frequent maintenance [96]. As software evolves, developers make changes to their code, test cases, system configuration, and dependencies, which may all lead to necessary changes in the build script. Adams et al. [47] found strong co-evolutionary relationship between source code and build script in their study. Since build scripts need to be synchronized with source code and the whole build environment, neglecting such changes in build scripts often leads to build failures.

According to our statistics on TravisTorrent [62] dataset on the continuous integration of open-source software projects, 29% of code commits fail to go through a successful build on the integration server. Seo *et al.* [178] also mentioned a similar build failure proportion at Google, which is 37%. These build failures hinders a project's development process so that they need to be fixed as soon as possible. However, many developers do not have the required expertise to repair build

scripts [172]. Therefore, automatic repair of build scripts can be desirable for software project managers and developers.

Automatic generation of software patches is an emerging technique, and has been addressed by multiple previous research efforts. For example, GenProg [89] and PAR [112] achieve promising result for automatic bug fixing. But these works are designed for repairing source code written in different programming languages. In contrast, repairing build scripts has its unique challenges. First, although the code similarity assumption (both GenProg and PAR are taking advantage of this assumption to fetch patch candidates from other portion of the project or other projects) still holds for build scripts, build-script repair often involves open knowledge that do not exist in the current project, such as a newly available version of a dependency or a build-tool plug-in (See Example 1). Second, unlike source code bugs, build failures does not have a test suite to facilitate fault localization [160] and to serve as the fitness function [84]. Third, while different programming languages share similar semantics (so that code patterns / templates can be adapted and reused), the semantics of build scripts are very different from normal programs, so we need to re-develop abstract fix templates for build scripts.

Example 1 Gradle Version Dependency Change *(puniverse/quasar: 2a45c6f)*

```
task wrapper(type: Wrapper) {
- gradleVersion = '1.11'
+ gradleVersion = '2.0'
}
```

On the other hand, there are also special opportunities we can take advantage of in the repair of build scripts. First, build failures often provide richer log information than normal test failures, and the build failure log can often be used to determine the reason and location of a build failure. Second, build scripts are programs in a specific domain, so it is possible to develop more specific build-fix templates (e.g., involving more domain-specific concepts such as versions, dependencies instead of general concepts like parameters, variables). Third, many build failures are related to the build tools and environments. These failures are not project-specific and may be recursive [184, 234] in different projects, so fix patterns can often be used beyond a project's boundary.

In this paper, we propose a novel approach, HireBuild, to generate patches for build scripts. Our insight is that, since many software projects use the same build tool (e.g., Gradle), similar build failures will result in similar build logs. Therefore, given a build failure, it is possible to use its build-failure log to locate similar build failures from a historical build-fix dataset, and adapt historical fixes for these new failures. Specifically, our technique consists of the following three phases. First, for a given build failure, based on build log similarity, we acquire a number of historically fixed build failures that have the most similar build logs. We refer to these build fixes as seed fixes. Second, from build-script diffs of the seed fixes, we extract a number of fix patterns based on our predefined fix-pattern templates for build scripts, and rank the patterns by their commonality among seed fixes. To generate build-script diffs, our approach uses an existing tool GumTree [83] which extracts changes of Java source code, XML, JavaScript code change. Third, we combine the patterns with information extracted from the build scripts and logs of the build failure to generate a ranked list of patches, which are applied to the build script until the build is successful.

Although following the general generation-validation process for program repair, our technique is featured with following major differences to address the challenges and take advantage of the opportunities in build-script repair.

- **Build log analysis.** Build logs contain a lot of information about the location and reason of build failures, and sometimes even provide solutions. Our build log analysis parses build logs and extracts information relevant to build failures. Furthermore, HireBuild measures the similarity of build logs based on extracted information.

- **Build-fix-pattern templates.** There are a number of common domain-specific operations in build scripts, such as including / excluding a dependency, updating version numbers, etc. In HireBuild, we developed build-fix-pattern templates to involve these common operations specific to software build process.

- **Build validation.** In build-script repair, without test cases, we need a new measurement to

validate generated patches. Specifically, we use the successful notification in the build log and the numbers of compiled source files to measure build successfulness.

In our work, we focus on repair of build scripts, so we do not consider compilation errors or unit-testing failures (although they also cause build failures) as they can be easily identified based on build logs and may be automatically repaired with existing bug repair techniques. Furthermore, we use Gradle (based on Groovy) as our targeted build tool as it is the most promising Java build tools now, and recent statistics [187] show that more than 50% of top GitHub apps have already switched to Gradle.

In our evaluation, we extracted 175 reproducible build fixes with corresponding build logs and build script changes from Travistorrent dataset [62] on February 8, 2017, the build fixes are from 54 different projects). To evaluate HireBuild, we use the earlier 135 build fixes as our training set, and 40 later actual build failures (chronologically 135 earlier and 40 later bug fixes among the 175 regardless of which project they belong to) as our evaluations set. Among these 40 build failures, we reproduced 24 build failures in our test environment. Empirical evaluation results show that our approach is able to generate a fix for 11 of the 24 reproduced build script failures which gives same build output as developers' original fix. Overall, our work presented in the paper makes the following contributions.

- A novel approach and tool to automatic patch generation for build scripts to resolve software build failures.
- A dataset of 175 build fixes which can serve as the basis and a benchmark for future research.
- An empirical evaluation of our approach on real-world build fixes.
- An Abstract-Syntax-Tree (AST) diff generation tool for Gradle build scripts, which potentially have more applications.

The remaining part of this paper is organized as follows. After presenting a motivation example of how build-script repair is different from source-code repair in Section 6.3, we describe the

design details of HireBuild in Section 6.4. Section 5.4 presents the evaluation of our approach, while Section 6.7 presents discussion of important issues. Related works and Conclusion will be discussed in Section 6.8 and Section 6.9, respectively.

5.2 Motivating Example

In this section, we introduce a real example from our dataset to illustrate how patch generation of build scripts is different from patch generation of source code. Example 12 shows a build failure and its corresponding patch where the upper part shows the most relevant snippet in the build-failure log and the lower part shows the code change to resolve the build failure. The project name and commit id are presented after the example title.

Example 2 A Gradle Build Failure and Patch *(puniverse/quasar: Build Failure Version:017fa18, Build Fix Version:509cd40)*

```
Could not resolve all dependencies for configuration ':quasar-galaxy:
   compile'.
> A conflict was found between the following modules:
   - org.slf4j:slf4j-api:1.7.10
   - org.slf4j:slf4j-api:1.7.7
```

```
compile ("co.paralleluniverse:galaxy:1.4") {
    ...
    exclude group: 'com.google.guava', module: 'guava'
+ exclude group:  "org.slf4j", module:  '*'
      }
```

In this build failure, the build-failure log complains that there are two conflicting versions of `slf4j` module, and the bug fix is to add an exclusion of the module in the compilation of `Galaxy` component. Although this build fix is just a one-line simple fix, it illustrates differences between source-code repair and build-script repair in the following aspects.

First, it is possible to find from existing scripts or past fixes that we need to perform an `exclude` operation, however, since `org.slf4j` never appears in the script (it is transitively referred and will be downloaded from Gradle central dependency repository at runtime), the string "org.slf4j" can be hard to generate, and enumerating all possible strings is not a feasible solution.

The string can actually be generated by comparing the build-failure log and available modules in Gradle central dependency repository, but this is very different from source-code patching where all variable names to be referred to are already defined in the code (in the case when a generated fix contains a newly declared variable, the variable can have any name as long as it does not conflict with existing names in the scope).

Second, in build-script repair, we are able to, and need to consider build-specific operations. For example, we should not simply deem `exclude` as an arbitrary method name, but needs to involve its semantics into fix-pattern templates, so that we know a module name will follow the `exclude` command.

Third, the build log information is very important in that it not only provides the name of conflicting dependency, but also provides the compilation task performed when build failure happens, which can largely help patch generation tool to locate the build failure and determine where to apply the patch.

5.3 Approach

The overall goal of HireBuild is to generate build-script patches that can be used to resolve build failures. HireBuild achieves these goals with three steps: (1) log similarity calculation to find similar historical build fixes as seed fixes, (2) extraction of build-fix patterns from seed fixes, and (3) generation and validation of concrete patches for build scripts. In the following subsections, we first introduce preliminary knowledge on Gradle, and then describe the three steps of HireBuild with more details in the following subsections.

5.3.1 Gradle Build Tool

Gradle is a general purpose build management system based on Groovy and Kotlin [37]. Gradle supports the automatic download and configuration of dependencies or other libraries. It supports Maven and Ivy repositories for retrieving these dependencies. This allows reusing the artifacts of existing build systems.

A Gradle build may consist of one or more build projects. A build project corresponds to the building of the whole software project or a submodule. Each build project consists of a number of tasks. A task represents a piece of work during the building process of the build project, e.g., compile the source code or generate the Javadoc. A project using Gradle describes its build process in the `build.gradle` file. This file is typically located in the root folder of the project. In this file, a developer can use a combination of declarative and imperative statements in Groovy or Kotlin code. This build file defines a project and its tasks, and tasks can also be created and extended dynamically at runtime. Gradle is a general purpose build system hence this build file can perform any task.

5.3.2 Log Similarity Calculation to Find Similar Fixes

One of the most important characteristic of build script repair is that, a lot of software projects use the same build tools (e.g., Gradle), so that build-failure logs of different projects and versions often share the same format and output similar error messages for similar build errors. So given a new build failure, HireBuild measures the similarity between its build-failure log and the build-failure logs of historical build failures to find its most similar build failures in history dataset.

Build Log Parsing

Gradle build logs typically contain thousands of lines of text. Gradle prints these lines when performing different tasks such as downloading dependencies, compiling source files, and when facing errors during the build. Our point of interest is the error-and-exception part, which typically accounts for only a small portion of the build log. So if we use the whole build log to calculate similarity, the remaining part will bring a lot of noises to the calculation (e.g., build logs from projects that have similar dependencies may be considered similar).

Therefore, we use only the error-and-exception part of the build log to calculate similarity between build logs. An example of the error-and-exception part in Gradle build log is presented as below.

```
* What went wrong:
A problem occurred evaluating project ':android-rest'.
>
Gradle version 1.9 is required. Current version is 1.8. If using the
   gradle wrapper, try editing the distributionUrl in /home/travis/
   build/47deg/appsly-android-rest/gradle/wrapper/gradle-wrapper.
   properties to gradle-1.9-all.zip
```

To extract the error-and-exception part, HireBuild extracts the portion of the build log after the error indicating header in Gradle (e.g., "* What went wrong"). HireBuild extracts only the last error, as the earlier ones are likely to be errors that are tolerated and are thus not likely to be the reason for the build failure. Furthermore, when there are exception stack traces in the error-and-exception part, HireBuild removes the stack traces for two reasons. First, stack traces are often very long, so they may dominate the main error message and bring noise (as mentioned above). Second, stack traces are often different from project to project so they cannot catch the commonality between build failures.

Text Processing

After we extracted the error-and-exception part from the build-failure log, we perform the following processing to convert the log text to standard word vector.

- `Text Normalization` breaks plain text into separate tokens and splits camel case words to multiple words.
- `Stop word Removal` removes common stop words, punctuation marks etc. For better similarity, HireBuild also removes common words for building process including "build", "failure", and "error".
- `Stemming` is the process of reducing inflected words to their root word. As an example, the word "goes" derived from word "go". The stemming process converts "goes" to its root

word “go”. For stemming we applied popular Porter stemming algorithm [214].

Similarity Calculation

With the generated word vector from the error-and-exception part of the build-failure logs, we use the standard Term Frequency–Inverse Document Frequency (TF–IDF) [170] formula to weight all the words. Finally, we calculate cosine similarity between the log of build failure to be resolved and all build-failure logs of historical build fixes in our training set, and fetch the most similar historical fixes. HireBuild uses the five most similar historical fixes as seed fixes to generate build-fix patterns.

5.3.3 Generation of Build-Fix Patterns

To generate build-fix patterns, for each seed fix, HireBuild first calculates the code difference between the versions before and after the fix. The code difference consists of a list of elementary revisions including insertions, deletions and updates. Then, for each revision, HireBuild generalizes it to hierarchical pattern and merges similar patterns. Finally, HireBuild flattens the hierarchical pattern to generate a set of build-fix patterns, and ranks these patterns.

Build-Script Differencing

In this phase, for each seed fix, we extract Gradle build script commits before and after fix, and convert the script code to AST representation. Gradle build uses Groovy [37]-based scripting language extended with domain-specific features to describe software build process. With support of the Groovy parser, AST representation of script code can be generated.

Our goal is to generate an abstract representation of code changes between two commits. Having build script content represented as an AST, we can apply tree difference algorithms, such as ChangeDistiller [86] or GumTree [83], to extract AST changes with sufficient abstraction. In particular, HireBuild uses GumTree to extract changes between two Gradle build scripts. GumTree generates a diff between two ASTs with list of actions which can be insertion, deletion, update, and movement of individual AST nodes to transfer from a source version to a destination version.

However, GumTree generates a list of AST revisions without node type information, so we revise GumTree to include the information. Furthermore, HireBuild also records the ancestor AST nodes of the changed AST subtree. Such ancestor AST nodes are typically the enclosing expression, statement, block, and task of the change, and they are helpful for merging different seed fixes for more general patterns, and for determining where the generated patches should be applied. As mentioned earlier, in Gradle scripts, a task is a piece of work which a build performs, and a script block is a method call with parameters as closure [38], so keeping such information helps to apply patches to a certain block or task. Example 3 shows an exemplar output of HireBuild's build-script differencing module, in which the operation, node type, and ancestor nodes are extracted. Note that HireBuild extracts only one level of parent expression to avoid potential noises. As shown in the example, the task/block name can be empty if the fix is not in any tasks/blocks.

Example 3 Build Script Differencing Output *(BuildCraft/BuildCraft: 98f7196)*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<patch>
<lineno id="30"><exp id="0">
   <operation>Update</operation>
   <nodetype>ConstantExpression</nodetype>
   <nodeexp>1.7.2-10.12.1.1079</nodeexp>
   <nodeparenttype>BinaryExpression</nodeparenttype>
   <nodeparentexp>(version = 1.7.2-10.12.1.1079)
   </nodeparentexp>
   <nodeblockname>minecraft</nodeblockname>
   <nodetaskname> </nodetaskname></exp>
  </lineno>
</patch>
```

Hierarchical Build-Fix Patterns

In some rare cases we can directly use the concrete build-fix pattern to generate a correct patch. Example 4 provides such a patch from `Project: nohana/Laevatein:a2aaca4`. There exists an exactly same build fix in the training set (from a different project).

However, in more common scenarios, code diffs generated from seed fixes are too specific and cannot be directly applied as patches. Consider Examples 5 and 6, changes made in different project are similar, but if we consider concrete change of Example 5 as "Update 1.7.2-10.12.1.1079"

Example 4 Training Project Fix *(journeyapps/zxing-android-embedded: 12cfa60)*

```
+ lintOptions {
+ abortOnError false
+ }
```

then this change can hardly be applied to other scripts. Therefore, we need to infer more general build-fix patterns from them.

Example 5 Gradle Build Fix *(BuildCraft/BuildCraft: 98f7196)*

```
- version = "1.7.2-10.12.1.1079"
+ version = "1.7.2-10.12.2.1121"
```

Example 6 Gradle Build Fix *(ForgeEssentials/ForgeEssentialsMain:fcbb468)*

```
-version = "1.4.0-beta7"
+version = "1.4.0-beta8"
```

Specifically, HireBuild infers a hierarchy of build-fix patterns from each seed fix by generalizing each element in the differencing output of the seed fix. For example, the hierarchies generalized from Examples 5 and 6 are shown in Figure 5.1. From the figure, we can see that, HireBuild does not generalize operations and the node type of expression that are involved in the fix (i.e., ConstantExpression), because a change on those typically indicates a totally different fix. HireBuild also does not include the task and block information in the pattern as they are typically not a part of the fix. Given a hierarchy, by choosing whether and which leaf node to be generalized, we can generate patches at different abstract levels. For example, if we generalize the parent expression from `version=1.7.2...` to `ParentExp: any`, we generate a pattern that updates a value `1.7.2...` without considering its parent. If we generalize both the parent expression and the node expression, we generate a pattern that update any constants in the script. Note that HireBuild does not consider the cases where the node expression is generalized but the parent expression is not, as such a pattern can never match real code.

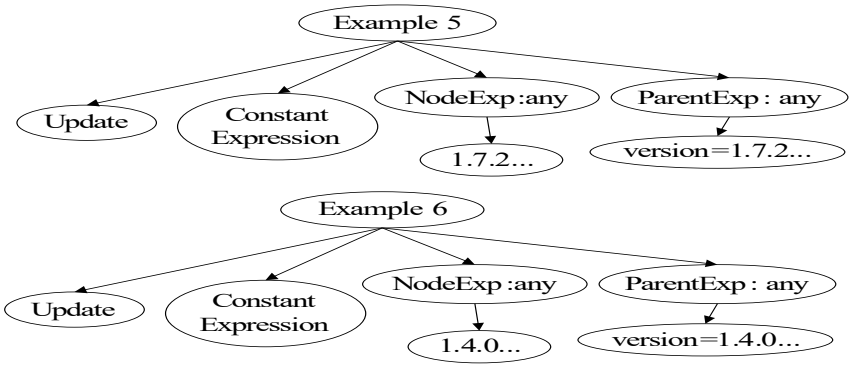


Figure 5.1: Hierarchies of Build-Fix Patterns

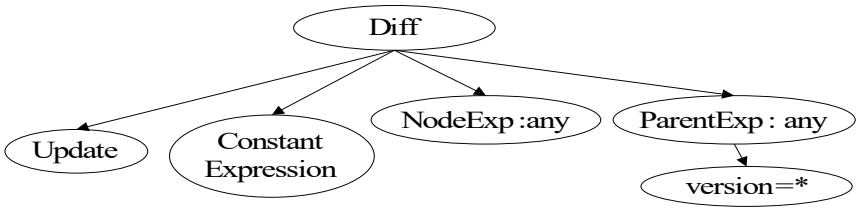


Figure 5.2: Merged Hierarchies

Merging Build-Fix Patterns

After generating hierarchies of build-fix patterns, HireBuild first tries to merge similar hierarchies. For example, the two hierarchies in Figure 5.1 will be merged to a hierarchy shown in Figure 5.2. HireBuild merges only a pair of hierarchies with the same operation and node type (Update and Constant Expression in this case). During the merging process, HireBuild merges hierarchies recursively from their root node, and merges nodes with exactly the same value. If two nodes to be merged have different constant values, HireBuild does not merge them and their children nodes. If two nodes to be merged have different expression values, HireBuild extracts their corresponding AST tree, and merges the AST tree so that the common part of the expressions can be extracted. In Figure 5.2, since the expressions `version=1.7.2...` and `version=1.4.0...` share the same child nodes `version` and =, a node `version=*` is added. Note that more than two hierarchies can be merged in the same way if they share the same operation and node type.

Ranking of Build-Fix Patterns

After hierarchies are merged, HireBuild calculates frequencies of build-fix patterns among seed fixes. If a hierarchy cannot be merged with other hierarchies, all the build-fix patterns in it have a frequency 1 among seed fixes, as they are specific to the seed fix they are from. In a merged hierarchy, the frequency of all its patterns is always the number of original hierarchies being merged. After calculating the frequencies of all build-fix patterns from hierarchies, HireBuild ranks build-fix patterns according to the frequency. For each build-fix pattern α, we counted n_α^t: α's frequency among seed fixes. Then probability of α is as follows.

$$P_\alpha = \frac{n_\alpha^t}{N}$$

where N is the total occurrences of build-fix patterns. Then, we rank the fix patterns based on the probability so that we use higher ranked build-fix patterns first to generate concrete patches. When there are ties between pattern A and B, if A is a generalization of B (A is generated by generalizing one or more leaf nodes of B), we rank B over A. The reason is that, when a build-fix pattern is generalized, it can lead to a larger number of concrete patches (e.g., update `gradleVersion` from any existing version to another existing version), so HireBuild needs to perform more build trials to exhaust all possibilities. As an example, all build-fix patterns from the hierarchy in Figure 5.2 have the same popularity, but the most concrete pattern: `update constant expression with parent expression version= *` will be ranked highest.

If there is no generalization relation between patterns, HireBuild ranks higher the build-fix patterns from the seed fix with higher ranking (the seed fix whose build failure log is more similar to that of the build failure to be fixed).

5.3.4 Generation and Validation of Concrete Patches

Before generation of concrete patches, we need to first decide which `.gradle` file to apply the fix. HireBuild uses a simple heuristic, which always choose the first `.gradle` file mentioned in

the error part extracted from the build failure log. If no `.gradle` file is mentioned, HireBuild uses the `build.gradle` file in the root folder.

Given a build-fix pattern, and the buggy Gradle build script as input, to generate concrete patches, HireBuild first parses the buggy Gradle build script to AST, and then HireBuild tries to find where a patch should be applied.

For updates and deletions, HireBuild matches the build-fix patterns to nodes in the AST. For example, the build-fix pattern `update constant expression with parent expression version = *` can be mapped to an AST node of type `ConstantExpression` and its parent expression node has a value matching `version=*`. When a build-fix pattern can be mapped to multiple AST nodes (very common for general build-fix patterns), and HireBuild generates patches for all the mapped AST nodes. The only exception is when a build-fix pattern is mapped to multiple AST nodes in one block. In build scripts, within the same block, the sequence of commands typically does not matter, so HireBuild retains only the first mapped node in the block to reduce duplication.

For insertions, it is impossible to map a build-fix pattern to an existing AST node, so HireBuild matches the block and task names of the build-fix patterns to the buggy build script. When a build-fix pattern is generated from a hierarchy merging multiple seed-fixes, HireBuild considers the task and block names of all seed-fixes. If a task or block name in the buggy script is matched, HireBuild inserts the build patch at the end of the task or block.

After HireBuild determines which build-fix pattern to apply and where to apply, we finally need to concentrate on the abstract parts of the build-fix pattern and determine the values of the abstract nodes (e.g., value of "*" in the pattern `update constant expression with parent expression version= *`). The most commonly used values in build scripts are (1) identifiers including task names, block names, variable names, etc.; (2) names of Gradle plug-ins and third-party tools / libraries; (3) file paths within the project; and (4) version numbers. HireBuild first determines which type the value to added belongs to, based on the concrete values and AST nodes in the seed fixes leading to the build-fix pattern to be applied. HireBuild identifies version

number and file paths based on regression expression matching (e.g., HireBuild can determine that `1.4.0-beta8` is a version number), and task / block / variable names by scanning the AST containing the seed fix. Other types of values including dependencies / plug-in names, and file paths are all specific to certain AST nodes so that they can be easily identified. Once the value type is determined, HireBuild generates values differently for different types as follows.

- Identifiers: HireBuild considers identifiers in the concrete seed fixes, as well as all available identifiers at the fix location.
- Names of plug-ins / libraries / tools: HireBuild considers names appearing in the concrete seed fixes, in the build failure log, and in the buggy build script.
- File paths: HireBuild considers paths appearing in the concrete seed fixes, in the build failure log, and in the buggy build script.
- Version numbers: HireBuild first locates the possible tools / libraries / plug-ins the version number is related to. This is done by searching for all occurrences of the version variable or constant in the AST of the buggy script. Once the tool / library / plug-in is determined, HireBuild searches Gradle central repository for all existing version numbers.

After the build-fix pattern, the location, and the concrete value are determined, a concrete patch is generated and added to the list of patches.

Ranking of Generated Patches

The previous steps generate a large number of patch candidates, so ranking of them is necessary to locate the actual fix as soon as possible. HireBuild ranks concrete patches with the following heuristics. Basically, we give higher priority to the patches which involve values or scopes more similar to the buggy script and the build-failure log.

1. Patches generated from higher ranked build-fix patterns are ranked higher than those generated from lower ranked build-fix patterns. The initial priority value of a patch is the probability value of its build-fix pattern.

2. If a patch p is to be applied to a location L, and p is generated from a build-fix pattern hierarchy merged from seed fixes A_1, ..., A_i, ..., A_n. If L resides in a task / block whose name is the same as the task / block name in any A_i, HireBuild adds p's priority value by 1.0.

3. If a patch involves a value (any one of the four types described in Section 5.3.4) which appears in the build-failure log. HireBuild adds the priority value of the patch by 1.0.

4. Rank all patches with updated priority values.

Note that, since the initial priority value is from 0 to 1, in the heuristics, we always add the priority value by 1.0 when certain condition meets, so that it go beyond all the other patches which do not satisfy the condition, no matter how high the initial priority value is.

Patch Application

After the ranked list of patches are generated, HireBuild applies the patches one by one until a timeout threshold is reached or the failure is fixed. HireBuild determines the failure is fixed if (1) the build process returns 0 and the build log shows build success, and (2) all source files that are compiled in the latest successfully built version are compiled if they are not deleted in between. We add the second criterion so that HireBuild can avoid trivial incorrect fixes such as changing the task to be performed from compile to clean up and to eliminate fake patches. HireBuild stops applying patches after it reaches the first patch passing the patch validation. Though there may be multiple valid patches, we apply only the first one that passes the validation.

HireBuild generally focuses on one line fixes as most other software repair tool does. But it also includes a technique to generate multi-line patches if the failure is not fixed until all single line patches are applied. Multi-line patches can be viewed as a combination of single line patches, but it is impossible to exhaust the whole combination space. Example 7 shows a bug fix, which can be viewed as the combination of three one-line patches (two deletions and one insertion). To reduce the search space of patch combination, HireBuild considers only the combination that occurs in

original seed fixes. Consider two one-line patches A and B, which are generated from hierarchies HA and HB. HireBuild considers the combination (A, B) only if HA and HB can be generalized from a same seed fix. After the filtering, HireBuild ranks patch combinations by the priority sum of the patches in the combination.

Example 7 Template with abstract node fix *(passy/Android-DirectoryChooser:27c194f)*

```
dependencies {
...
- testCompile files('testlibs/robolectric-2.4-SNAPSHOT-jar-
    with-dependencies.jar')
- androidTestProvided files('testlibs/robolectric-2.4-SNAPS
    HOT-jar-with-dependencies.jar')
+ androidTestCompile 'org.robolectric:robolectric:2.3+'
...
}
```

5.4 Empirical Evaluation

In this section, we describe our dataset construction in Section 5.4.1 and our experimental settings in Section 5.4.2, followed by research questions in Section 5.4.3 and experiment results in Section 5.4.4. Finally, we discuss the threats to validity in Section 5.4.5.

5.4.1 Dataset

We evaluate our approach to build-script repair on a dataset of build fixes extracted from the TravisTorrent dataset [62] snapshot at February 8, 2017. The tool and bug set used in our evaluation are all available at our website [1]. TravisTorrent provides easy-to-use Travis CI build data to the masses through its open database. Though it provides large amount of build logs and relevant data, our point of interest is build status transition from error or fail status to pass status with changes in build scripts. From the version history of all projects in the TravisTorrent dataset, we identified as build fixes the code commits that satisfy: (1) the build status of their immediate previous version is fail / error; (2) the build status of the committed version is success; and (3)

[1] HireBuild Dataset and Tools: https://sites.google.com/site/buildfix2017/

they contain only changes in gradle build scripts. Since HireBuild focuses on build script errors, we use code commits with only build-script changes so that we can filter out unit test failures and compilation failures. Our dataset may miss the more complicated build fixes that involve a combination of source-code changes and build-script changes, or a combination of build-script changes from different build tools (e.g., Gradle and Maven). HireBuild currently does not support the generation of such build fixes cross programming languages. Actually, fixing such bugs are very challenging and is not supported by any existing software repair tools.

From the commit history of all projects, we extracted a dataset of 175 build fixes. More detailed information about our data set is presented in Table 6.3. We can see that these fixes are from 54 different projects, with maximal number of fixes in one project to be 25.

We ordered the build fixes according to the code commit time stamp, and use 135 (75%) earlier build fixes as the training set and the rest 40 build fixes (25%) as the evaluation set. **Therefore, all the build fixes in our evaluation set are chronically later than the build fixes in our training set.** Note that we combine all the projects in both training sets and evaluation sets, so our evaluation is cross-project in nature.

Among these 40 build fixes for evaluation, we successfully reproduced 24 build failures. The remaining 16 build failures cannot be reproduced in our test environment for the following three reasons: (1) a missing library or build configuration file was originally missing from the central repository and caused the build failure, but they are added later; (2) a flawed third-party library or build configuration caused the build failure, but the flaws are fixed and flawed releases are no longer available on the Internet; and (3) the failure can be reproduced only with specific build commands and options which are not recorded in the repository. For case (3), we were able to reproduce some bugs by trying common build command options. We also contacted the TravisCI people about the availability of such commands / options, but they could not provide them to us. For training set, we did not reproduce build failures since we trust the software version history in TravisTorrent that human made changes resolved the build failures and we extracted only seed fixes from training set.

Table 5.1: Dataset Summary

Type	Count
# Total Number of Projects	54
# Maximum Number of Fix From Single Project	25
# Minimum Number of Fix From Single Project	1
# Average Number of Fix Per Project	3.2
# Total Number of Fix	175
# Training Fix Size	135
# Testing Fix Size	40
# Reproducible Build Failure Size for Testing	24

5.4.2 Experiment Settings

TravisTorrent datset provides Travis CI build analysis result as SQL dump and CSV format. We use SQL dump file for our experiment. We use a computer with 2.4 GHz Intel Core i7 CPU with 16GB of Memory, and Ubuntu 14.10 LTS operating system. We use MySQL Server 5.7 to store build fix changes. In our evaluation, we use 600 minutes as the time out threshold for HireBuild.

5.4.3 Research Questions

In our research experiment, we seek to answer following research questions.

- **RQ1** How many reproducible build failures in the evaluation set can HireBuild fix?
- **RQ2** What are the amount of time HireBuild spends to fix a build failure?
- **RQ3** What are the sizes of build fixes that can be successfully fixed and that cannot be fixed?
- **RQ4** What are the reasons behind unsuccessful build-script repair?

5.4.4 Results

RQ1: Number of successfully fixed build failures. In our evaluation, we consider a fix to be correct only if there is no build failure message in build log after applying patch, and the build result (i.e., all compiled classes) are exactly the same as those generated by the manual fix. Among 24 reproducible build failures in the test set, we can generate the correct fix for 11 of them. Table

Table 5.2: Project-wise Build Failure / Fix List

Project Name	#Failures	#Correctly Fixed
aol/micro-server	2	1
BuildCraft/BuildCraft	2	0
exteso/alf.io	1	1
facebook/rebound	1	1
griffon/griffon	1	0
/btrace	1	1
jMonkeyEngine/jmonkeyengine	2	0
jphp-compiler/jphp	1	0
Netflix/Hystrix	2	0
puniverse/quasar	6	2
RS485/LogisticsPipes	5	5
Total	24	11

5.2 shows the list of projects that are used for testing. Columns 2 and 3 represent the number of build failures and the number of those fixed successfully.

Figure 5.3 shows the breakdown of successful build fixes according the type of changes. With HireBuild, we can correctly generate 3 fixes about gradle option changes, 3 fixes about property changes, 2 fixes about dependency changes and external-tool option changes, respectively, and 1 fix about removing incompatible statements. Example 8 shows a build fix that is correctly generated by HireBuild falling in the category of external-tool option changes. The build failure is caused by adding a new option which is compatible only with Java 8. So the fix is to add an if condition to check the Java version. Note that this fix involves applying the combination of two insertion patches, but HireBuild still can fix it as there are a seed fix that contains both build-fix patterns.

Example 8 A Build Fix Correctly Generated By HireBuild *(puniverse/quasar:33bb265)*

```
+if(JavaVersion.current().isJava8Compatible()){
 +tasks.withType(Javadoc) {
 options.addStringOption('Xdoclint:none', '-quiet')
+ }
+ }
```

RQ2: Time Spent on Fixes Time spent on fixes is very important for build failures as they need to be fixed timely. The size of patch list has impact on automatic build script repair. If patch list size is too large, it will take large time span to generate fix sequence. We compare patch list

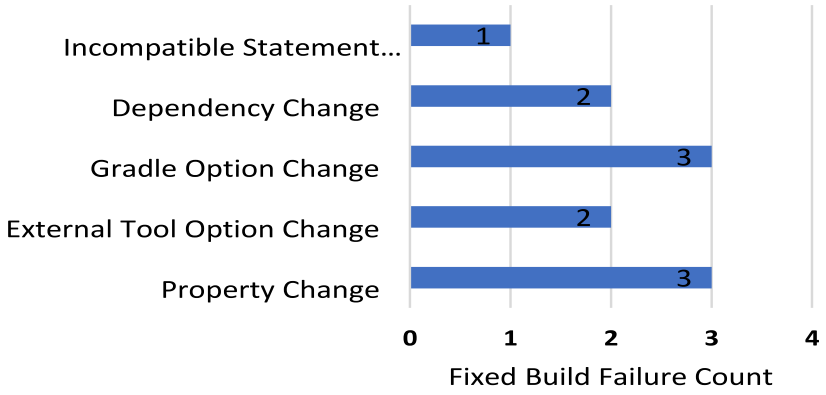


Figure 5.3: Breakdown of Build Fixes

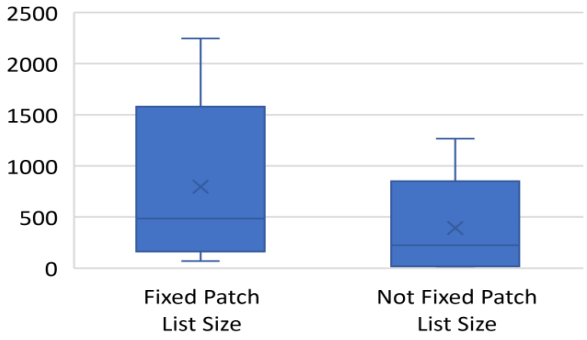


Figure 5.4: Patch List Sizes

size of build failures we can correctly fix and patch list size of build failures we cannot correctly fix, and present the result in Figure 5.4. From the figure, we can see that for fixed build failures, the patch list has minimum size of 68 and maximum size of 2,245, while median is 486. For non-fixed build failures, patch list minimum, median and maximum are 8, 223 and 1,266 respectively, which are lower than fixable build failure's patch list. The reason behind this result is that for non-fixable build failure, HireBuild cannot find similar build fixes in the training set, and thus the generated build-fix patterns cannot be easily mapped to AST nodes in buggy scripts.

For the 11 fixed build failures, we compared in Figure 5.5 the time HireBuild spent on automatic fixing build failures with the manual fix time of the build failures in the commit history. Manual fix time comes from commit information and we use the time difference between build failure inducing commit and build failure fix commit as the manual fix time. From Figure 5.5, we

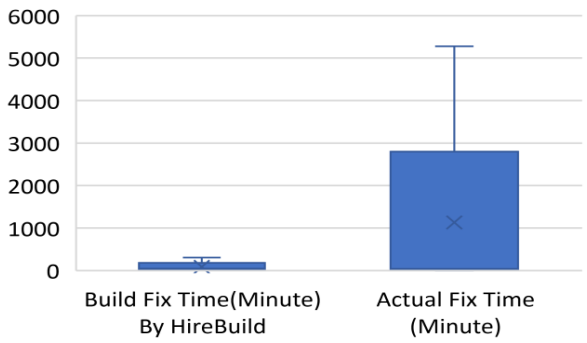


Figure 5.5: Amount of Time Required for Build Script Fix

can see that build script fix generated by our approach takes minimum 2 minutes, maximum 305 minutes and median value of 44 minutes. While human fix takes minimum less than one minute, maximum 5,281 minutes and median 42 minutes. We can see that for the fixable build failures, HireBuild fixed them with time comparable to manual fixes.

RQ3: Actual Fix Size. Patch size has impact on automatic program repair. According to Le et al. [120], bugs with over six lines of fix are difficult for automatic repair. In our dataset we have not performed any filtering based on actual fix size. But during result analysis, we performed statistical analysis to find out the sizes of build-fix patches, and the difference in size between the patches our approach can correctly generate and the patches our approach cannot. According to Figure 5.6, fixed build script failures contain minimum one, maximum two and median one. Actually 9 of the fixes contain only 1 statement change, and 2 of the fixes contain only 2 statement changes. For non-fixed Gradle build script failures minimum change size is one, while maximum and median change size is 11 and 1 respectively. Therefore, our approach mainly works in the cases where the number of statement changes is small (1 or 2), which is similar to other automatic repair tools.

RQ4: Failing reasons for the rest 13 build failures. For 54.16% of evaluated build failures, our approach cannot generate build fix. So, we performed manual analysis to find out why our approach fails. We check whether the reason is related to generation of version numbers, dependency names, etc. Then we categorize these failure reasons to four major groups: (1) Project specific change adaption, (2) Non-matching patterns, (3) Dependency resolution failures, and (4)

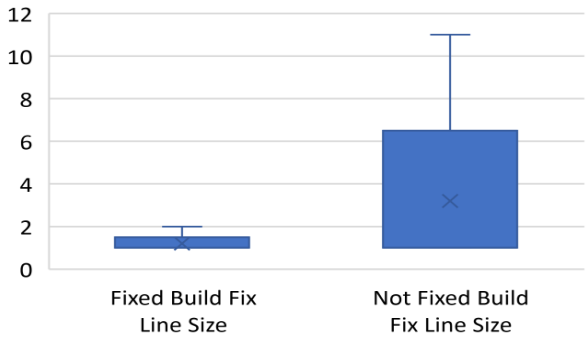


Figure 5.6: Actual Fix Sizes

Table 5.3: Cause of unsuccessful patch generation

Fix Type	#of Failures
Project specific change adaption	2(15%)
No matching patterns	6(46%)
Dependency resolution failures	3(23%)
Multi-location fixes	2(15%)

Multi-location fixes, as shown in Table 5.3.

Project specific change adaption indicates those changes that are dependent on project structure, file path etc. As build script manages build and its configuration, so there are project specific change issues and with our approach we cannot adapt the build-fix patterns. Example 9 shows such a build fix where it uses a specific path in build script.

Example 9 Project specific change *(Netflix/Hystrix:6600947)*

```
+ if ( dep.moduleName == 'servlet-api' ) {
+ it.artifactId[0].text() == dep.moduleName &&
+ asNode().dependencies[0].dependency.find{
+ ...
+ }}
+ }
```

Non-matching patterns indicates that our automatic patterns generation failed to provide required pattern that can resolve the build failure. HireBuild could not generate appropriate patterns for 6 failures which account 46% of failures. This may be due to limited size of training data and insufficient number of available build fixes that we used for template generation.

Dependency resolution failures happens for some project when HireBuild did not find an actual dependency from central repository based on build log error. Even if we find dependency based on miss-compiled classes, that may not match with actual fixing. Example 10 shows such a dependency update where our approach failed to generate the dependency name.

Example 10 Dependency resolve Issue *(BuildCraft/BuildCraft:12f4f06)*

```
- mappings = 'snapshot_20160214'
+ mappings = 'stable_22'
```

Multi-Location Fixes happen when we need to apply multiple patches to fix a single build failure. HireBuild considers only limited combinations of patches as introduced in Section 5.3.4. Example 11 shows such a case where our patch generation technique generated the two "exclude" statements in two different patches. But this build failure is fixed only when we apply both "exclude" statement change simultaneously.

Example 11 Dependency resolve Issue *(joansmith/quasar:64e42ef)*

```
-jvmArgs ' -Xbootclasspathp:
  ${System.getProperty(üser.home)}jsr166.jar'
-testCompile 'org.testng:testng:6.9.6'
+testCompile('org.testng:testng:6.9.6') {
 +exclude group:'com.google.guava', module:'*'
 +exclude group:'junit', module:'*'
 +}
```

5.4.5 Threats of Validity

There are three major threats to the internal validity of our evaluation. First, there can be mistakes in our data processing and bugs in the implementation of HireBuild. To reduce this threat, we carefully double checked all the code and data in our evaluation. Second, the successful fixes generated by HireBuild may still have subtle differences with the manual fixes. Furthermore, the manual fixes that we use as the ground truth may in itself has flaws. To reduce this threat, we used a strict criterion for correct fixes. We need the automatically generated fix to generate

exactly the same build results as those generated by the manual fix. Third, the manual fixing time collected in the commit history may be longer the actual fixing time as developers may choose to wait and not to fix the bug. We agree that this can happen but we believe the difference is not large as developers typically want to fix failure as soon as possible so that it does not affect other developers.

The major threat to the external validity of our evaluation is that we use a evaluation set with limited number of reproducible bug fixes. Furthermore, our evaluation set contains only build fixes where only Gradle build scripts are changed. So it is possible that our conclusion is limited to the data set, Gradle-script fixes, or Gradle-script-only fixes. Figure 5.3 shows that our evaluation set already covers a large range of change types, and we plan to expand our evaluation set to more build failures and reproduce more bugs as TravisTorrent data set grows overtime. Gradle is the most widely adopted building system now, and its market share is still increasing. We also did a statistics on the number of build fixes with both Gradle-script changes and other file-changes and found 263 of them. Compared with 175 build fixes in our dataset, Gradle-script-only fixes accounts for a large portion of build script fixes for Gradle systems.

5.5 Discussion

Patch Validation and Build Correctness. In the patch validation stage, HireBuild deems a patch as valid if applying it results in a successful build message, and all the files that are compiled in the most recent successful build are still compiled as long as they are not deleted. Our evaluation uses a more strict constraint which requires the compiled files in the automatically fixed build to be exactly the same as those in the manually fixed build. The evaluation results show that our patch validation strategy is very effective because in all 11 fixed builds in our evaluation, the first patches passing validation are confirmed to be correct patches. The reason is that, based on the same compiler, once a source file is successfully compiled, it is unlikely to be compiled in different ways. The only exception is that a library-class reference is resolved to a wrong class when a wrong dependency is added. Furthermore, to pass compilation, the wrong class must accidentally

have compatible behaviors (e.g., methods) with the correct class. Such coincidence is not likely to happen.

Build Environment. Build environment defines the environment of a system that compiles source code, links module and generates assembles. From developer's point of view, they install all required dependencies like Java, GCC and other frameworks. But when projects are built in different environment then build problem can be generated. For example, if certain project has dependency on Java 1.8 then building the project in build environment with Java 1.7 might generate build failure. This a challenge for build automation as well as automatic build repair. During software evaluation, developers change environment dependency based on functional requirements or efficiency. With version changes, developers build the software having those changed dependencies. But for build script repair, if we change the version of any dependency and keep the build environment as it was before, then the fix might not resolve build failures. For Android projects environment, this issue creates greater impact as in most Gradle build script it mentions SDK version, build tool version etc. inside build script. As a result, build script version dependency and build environment should be synced to avoid build breakage.

5.6 Related Work

5.6.1 Automatic Program Repair

Automatic program repair is gaining research interest in the software engineering community with the focus to reduce bug fixing time and effort. Recent advancements in program analysis, synthesis, and machine learning have made automatic program repair a promising direction. Early software repair techniques are mainly specific to predefined problems [111, 181, 204, 231]. Le Goues et al. [89] GenProg which is one of the earliest and promising search based automatic patch generation technique based on genetic programming. Patch generated by this approach follows random mutation and use test case for the verification of the patch. Later in 2012, authors optimized their mutation operation and performed systematic evaluation of 105 real bugs [89]. RSRepair [164] performs similar patch generation based on random search. D. Kim et al. [112] proposed an ap-

proach to automatic software patching by learning from common bug-fixing patterns in software version history, and later studied the usefulness of generated patches [191]. AE [209] uses deterministic search technique to generate patch. Pattern-based Automatic Program Repair(PAR) [112] uses manually generated templates learned from human written patches to prepare a patch. PAR also used randomized technique to apply the fix patches. Nguyen et al. [152] proposed SemFix, which applied software synthesis to automatic program repair, by checking whether a suspicious statement can be re-written to make a failed test case pass. Le et al. [120] mines bug fix patterns for automatic template generation and uses version control history to perform mutation. Prophet [128] proposed a probabilistic model learned from human written patched to generate new patch. The above mentioned approaches infers a hypothesis that new patch can be constructed based on existing source. This hypothesis also validated by Barr et al. [56] that 43 percent changes can be generated from existing code. With this hypothesis, we proposed first approach for automatic build failure patch generation. Tan and Roychoudhury proposed Relifix [189], a technique that taking advantage of version history information to repair regression faults. Smith et al. [183] reported an empirical study on the overfitting to test suites of automatically generated software patches. Most recently, Long and Rinard proposed SPR [127], which generates patching rules with condition synthesis, and searches for the valid patch in the patch candidates generated with the rules. Angelix [144] and DirectFix [143] both use semantics-based approach for patch generation. To fix buggy conditions, Nopol [220] proposes test-suite based repair technique using Satisfiability Modulo Theory(SMT). Although our fundamental goal is same, but our approach is different than others in several aspects: 1) Our approach is applicable for build scripts, 2) We generate automatic fix template using build failure log similarity, 3) With abstract fix template matching we can generate fix candidate lists with reasonable size.

5.6.2 Analysis of Build Configuration Files

Analysis of build configuration files is growing as an important aspect for software engineering research such as dependency analysis for path expression, migration of build configuration file

and empirical studies. On dependency analysis, Gunter [51] proposed a Petri-net based model to describe the dependencies in build configuration files. Adams et al. [46] proposed a framework to extract a dependency graph for build configuration files, and provide automatic tools to keep consistency during revision. Most recently, Al-Kofahi et al. [48] proposed a fault localization approach for make files, which provides the suspiciousness scores of each statement in a make files for a building error. Wolf et al. proposed an approach [215] to predict build errors from the social relationship among developers. McIntosh et al. [140] carried out an empirical study on the efforts developers spend on the building configurations of projects. Downs et al. [79] proposed an approach to remind developers in a development team about the building status of the project. On the study of building errors, Seo et al. [178] and Hassan et al. [94, 97] carried out empirical studies to categorize build errors. Their study shows that missing types and incompatibility are the most common type of build errors, which are consistent with our findings.

The most closely related work in this category is SYMake developed by Tamrawi et al. [188]. SYMake uses a symbolic-evaluation-based technique to generate a string dependency graph for the string variables/constants in a Makefile, automatically traces these values in maintenance tasks (e.g.,renaming), and detect common errors. Compared to SYMake, the proposed project plans to develop build configuration analysis for a different purpose (i.e., automatic software building). Therefore, the proposed analysis estimates run-time values of string variables with grammar-based string analysis instead of string dependency analysis, and analyzes flows of files to identify the paths to put downloaded files and source files to be involved. On migration of build configuration files, AutoConf [12] is a GNU software that automatically generates configuration scripts based on detected features of a computer system. AutoConf detects existing features (e.g., libraries, software installed) in a build environment, and configure the software based on pre-defined options.

5.7 Conclusion And Future Work

For Source code, automatic patch generation research is already in good shape. Unfortunately, existing techniques are only concentrated to source code related bug fixing. In this work, we

propose the first approach for automatic build fix candidate patch generation for Gradle build script. Our solution works on automatic build fix template generation based on build failure log similarity and historical build script fixes. For extracting build script changes, we developed GradleDiff for AST level build script change identification. Based on automated fix template we generated a ranked list of patches. In our evaluation, our approach can fix 11 out of 24 reproducible build failures.

In future, we plan to increase training and testing data size for better coverage of build failures with better evaluation and perform study on patch quality for the patches generated by out tool. Moreover, change patterns from general build script commits may also be useful, and we have a plan to work on build script change patterns regardless of build status. Apart from that, we are planning to apply search based technique such as genetic programming with fitness function on our patch list to better rank our generated patches and apply combination of patches.

ACKNOWLEDGEMENTS. The authors are supported in part by NSF Awards CCF-1464425, CNS-1748109, and DHS grant DHS-14-ST-062-001.

CHAPTER 6: UNILOC: UNIFIED FAULT LOCALIZATION OF CONTINUOUS INTEGRATION FAILURES

This chapter presents UniLoc, a unified technique to localize faults in both source code and build scripts given CI failures. 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. A significant portion of this work has been under submission at the following journal:

- Foyzul Hassan, Na Meng, Xiaoyin Wang, "UniLoc: Unified Fault Localization of Continuous Integration Failures," in submission at IEEE Transactions on Software Engineering (TSE).

6.1 Introduction

As an emerging software engineering practice [57], Continuous Integration (CI) [80] enables developers to identify integration errors in earlier phases of software process, significantly reducing project risk and development cost. Meanwhile, *CI poses high demands for efficient fault localization and program repair techniques to ensure 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 [196]. 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 [42], and diminish people's confidence in adopting CI [43]. Existing fault localization (FL) techniques either rely on bug reports or test failures to locate bugs in source code [75, 108, 221, 236]. However, CI failures bring new challenges to these techniques.

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 [172] 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: Build failures. 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. [172] 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 ***build failures***. Existing approaches are unable to localize bugs for build 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. UniLoc takes both source files and build scripts into consideration, and conducts unified fault localization to diagnose both test failures and build failures. 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 [177, 211], UniLoc also extracts Abstract Syntax Trees (ASTs) from source files and build files to divide large documents into smaller ones. However, prior 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 evaluate UniLoc, we collected 700 real CI failure fixes in 63 GitHub projects from the TravisTorrent dataset [62]. We used earlier 100 fixes for parameter tuning, and the remaining 600 fixes for evaluation. As with prior work [75, 177, 236], we evaluated UniLoc's effectiveness by measuring Top-N, MRR, and MAP. 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 and MAP values of UniLoc were 0.49 and 0.36.

This paper is the first work on unified fault localization of both test failures and build failures. To compare UniLoc with existing code-oriented IRFL, we applied BLUiR [177] to the 600 bug fixes because BLUiR is a widely used IRFL approach [75, 122]. The MRR and MAP values of BLUiR were 0.29 and 0.19, much lower than those values of 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 conducted a comprehensive evaluation with real CI failures on UniLoc and BLUiR. We explored how UniLoc works differently from code-oriented IRFL techniques. We also investigated how different optimization strategies affect the effectiveness of UniLoc.

- We constructed a data set of 700 CI failure fixes together with the related failure-inducing commits. We open sourced the data to facilitate future research in CI failure repair.

The organization of the paper is as follows. After describing the background of this work in Section 6.2, we introduce UniLoc in Section 6.4. Section 6.5 explains evaluation and Section 6.7 discusses the generalization of our approach. We expound on the related works and conclusion in Section 6.8 and Section 6.9, respectively.

6.2 Background

This section first clarifies terminology (Section 6.2.1), and then introduces the IR technique we used (Section 6.2.2). Finally, it explains the project dependencies manifested by build scripts (Section 6.2.3).

6.2.1 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.

6.2.2 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 a descending order of the scores. Below is a frequently used formula for similarity calculation:

$$Sim(q, d) = cos(q, d) = \frac{\overrightarrow{V_q}.\overrightarrow{V_d}}{\mid \overrightarrow{V_q} \mid\mid \overrightarrow{V_d} \mid} \tag{6.1}$$

$\overrightarrow{V_q}$ 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.

6.2.3 Project Dependencies

A build system is an infrastructure to convert source code into artifacts such as modules, libraries, and executable binaries [141]. 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 and retested. Instead, the build system only compile and test the sub-projects being changed and those sub-projects depending on the changed ones.

Figure 6.1 presents three Gradle scripts defined in the project *spockframework/spock*. In Figure 6.1 (a), script *settings.gradle* shows that the project includes multiple sub-projects, such as *spock-bom* and *spock-core*. In Figure 6.1 (b), `compile/testcompile project(":spock-core")` means that *spock-core* is needed for Gradle to compile the owner sub-project. Figure 6.1 (c) indicates that *spock-core* is needed to compile and test *spock-specs*.

```
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

Figure 6.1: Three *.gradle files used in *spockframework/spock* declare sub-projects and specify dependencies between the projects

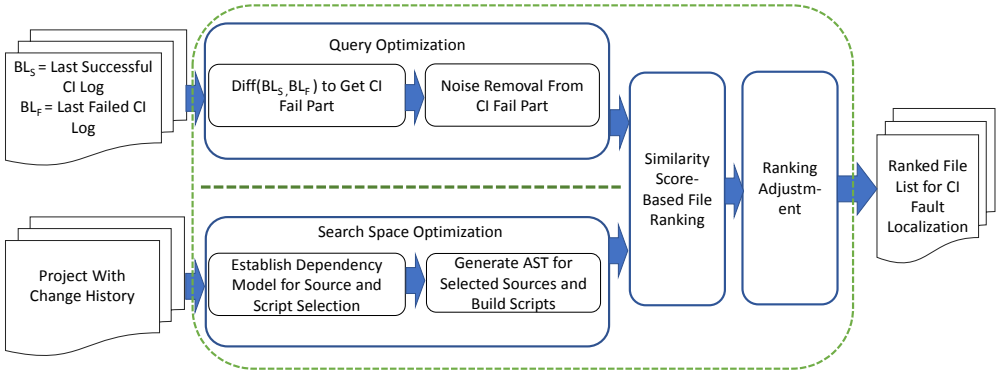


Figure 6.2: UniLoc Architecture

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.

6.3 Motivating Example

This section motivates our approach design with a real CI failed log and the related buggy file. In Example 12, the upper part shows a log snippet describing a compilation error, while the lower part presents the buggy file that was later fixed. Comparing the two parts, we observed that the failed log and buggy file share some unique words in common, such as `IRCFragment` and `getConversation`. This observation inspired us to take an IR-based approach for fault localization, because (1) buggy files are likely to share words with the failed log and (2) IR techniques are good at retrieving and ranking documents for queries based on the commonly used words.

Example 12 A CI failed log and the corresponding buggy Java file *(CI Failure Version: fe82cc1, Fix Version: 4d3bf5d)*

```
~/ServerFragment.java:37: error: ServerFragment is not abstract
   and does not override abstract method getConversation() in
   IRCFragment
public class ServerFragment extends IRCFragment;ServerEvent; {
     ^
...
FAILED
```

```
Source code file: IRCFragment.java
Commit ID: fe82cc1
abstract class IRCFragment<T extends Event> extends ListFragment
   implements TextView.OnEditorActionListener {
       ...
       public abstract Conversation getConversation();
       ...
}
```

6.4 Approach

Figure 6.2 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 commit (file changes) C_f between the two versions. 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 6.4.1).
- 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 6.4.2).

- Phase 3 compares each document within the scope against q to calculate the similarity score and rank documents accordingly (Section 6.4.3).
- Phase 4 retrieves C_f, the failure-inducing commit, and extracts the file-level change information to improve ranking formula (Section 6.4.4).

6.4.1 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 the unrelated information into 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 6.4.1), and (ii) noise removal with text similarity (Section 6.4.1).

Query Optimization with Text Diff

We observed that a failed log can 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 [173], we applied a textual differentiation algorithm—Myers [150]—to BL_f and BL_s to identify any failure-related part in BL_f. Myers finds the longest common subsequence of two given strings. Example 13 shows a passed log (commit:0545247) and a failed log (commit:bf25fdf) with the unique fragments in the failed log highlighted with red. UniLoc can extract these fragments using Myers.

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

Example 13 CI Log Diff *(BuildCraft/BuildCraft)*

```
CI Log Part for Commit ID: 0545247
...
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
...
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.
```

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, even though such differences are irrelevant to failures.

To further remove such noises, we leveraged an intuition that the failure-irrelevant lines in $PART_f$ should be similar to the lines in $PART_s$. In particular, we compared each line from $PART_f$ (e.g., l_f) with each line of $PART_s$ (e.g., l_s) using Myers. If the similarity between l_f and l_s is above a threshold lt, we consider the lines to match and l_f should be removed from $PART_f$. To decide the optimum value of lt, we used 100 CI failure fixes in our dataset (see Section 6.5.1) 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.

6.4.2 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.

Dependency-Based Sub-Project Selection

As described in Section 6.2.3, 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

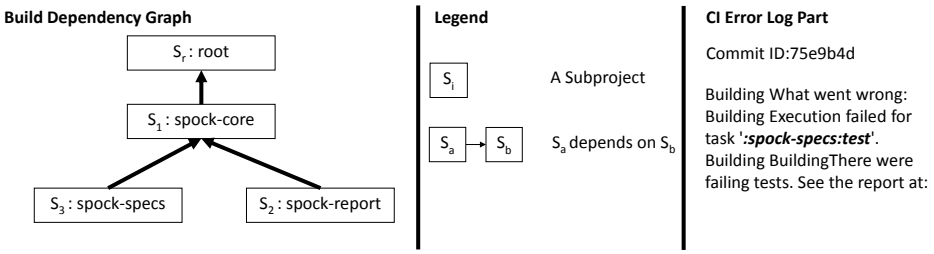


Figure 6.3: Sub-Project Dependency Graph

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 6.3 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`.

AST-Based Entity Extraction

Prior work [177] 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 [177] 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 search for the textual values in its build scripts and build failure logs. Finally, we count 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 [177]) AST node types with the highest frequency as top software

Table 6.1: Top Software Entities Source Code

AST Node	Frequency	Example
Field 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 6.2: 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)**

entities. Note that this is the same tuning set we used for query noise removal (See Section 6.4.1). In Tables 6.1 and 6.2, we show the statistic 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 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 6.1 and 6.2, *field names*, *method names*, *class names*, and *import items* are top four software entities in Java source code; *dependency items*, *property definitions*, *task names*, and *module names* are top four software entities in Grade build scripts. Therefore, we include only textual values of them in the search space.

6.4.3 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 languaged 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 [24] for the IR technique described in Section 6.2.2. Given $PART'_f$ and refined search scope, this implementation creates a query vector q and a set of document vectors D.For source code and build script, we perform separate search for each document field and calculate average of the search score as similarity. Saha et al. [177] also performed similar approach but due to homogeneous nature of their fault localization they used sum of similarity score. Since we are considering source code and build script and their used fields are different, we used average of search score rather than sum of score. The higher score a document has, the higher it is ranked.

$$Sim(q, d) = \frac{1}{N_f} \sum_{f \in D} Sim(q, d_f) \tag{6.2}$$

Here f is particular field for each source code and build script and N_f denotes number of field. After calculating similarity with each document field with query q, we calculate average value for similarity score.

6.4.4 Ranking Adjustment

To further improve file ranking, we leveraged an intuition that a changed file in a failure-inducing commit and 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 consider the commit C_f between V_p and V_f to be the failure-inducing commit and considered the changed files in C_f and files mentioned in $PART'_f$ as suspicious files. We defined the following formula to adjust similarity scores for documents:

$$FinalScore(x) = \begin{cases} Score(q,d)^{\alpha}, & \text{if } d \text{ is a suspicion file} \\ Score(q,d), & \text{otherwise} \end{cases} \tag{6.3}$$

In Equation 6.3, if a file is changed in any failure-inducing commit or mentioned in $PART'_f$, we raised 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. [223]'s prior work, they defined a formula to boost suspicious score when certain files are recently changed, and we were inspired by their formula.

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 6.4.1. The experiments showed that 0.1 is the best setting, so we set $\alpha = 0.1$ by default.

6.5 Experiments and Analysis

In this section, we will first introduce our dataset (Section 6.5.1) and evaluation metrics (Section 6.5.2). We will then describe the research questions we explored (Section 6.5.3) and finally discuss the evaluation results (Section 6.5.4).

6.5.1 Datatset

We constructed our evaluation dataset based on TravisTorrent [62], which is a dataset of CI builds. We used the SQL dump version of the dataset dated February 8, 2017 snapshot with 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 [187] 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

Table 6.3: Dataset Summary

Type	Count
# Total Number of Projects	72
# 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
# Tuning Set Size	100
# Evaluation Set Size	600

each CI failure, we further extracted the most recent 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 [119, 169, 200] to use all changes in the fixing commit as the ground truth.

In this way, we identified 700 CI failures fixes from 72 Gradle-based projects. As shown in Table 6.3, we used the chronologically earliest 100 of the fixes for parameter tuning (see Section 6.4.1 and Section 6.4.4) and software entity selection of Java source code and Gradle build script (see Section 6.4.2). We used the remaining 600 fixes to evaluate the effectiveness of our fault localization techniques[1]. .

More importantly, we manually inspected the 700 failed logs and their corresponding fixes. We clustered the data based on (1) the failure type and (2) bug locations. As shown in Table 6.4, 316 (45%) CI failures are test failure and 384 (55%) failures are due to build failures. 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.

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 the current FL tool does

[1]Our dataset is available at https://sites.google.com/view/uniloc

Table 6.4: Failure Types and Bug Locations

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

not analyze build scripts to locate bugs, they can miss bug locations for many CI failures. In particular, Example 14 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 15 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.

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

Example 14 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 15 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
```

6.5.2 Evaluation Metrics

We used the following three widely used metrics [75, 169, 217, 224, 236] to measure the effectiveness of FL techniques.

- **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:

$$RR = \frac{1}{rank_{best}} \tag{6.4}$$

  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:

$$AP = \sum_{k=1}^{M} \frac{P(k) * pos(k)}{number\ of\ positive\ instances} \tag{6.5}$$

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 $(\frac{1}{3} + \frac{2}{5})/2 = 0.37$.

6.5.3 Research Questions

In our experiment, we investigated following five research questions:

- **RQ1** How effective is UniLoc to locate bugs for CI failures? 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 a baseline IR-based technique to the same data set.

- **RQ2** What is the impact of recent change based file ranking for build fault localization? To answer RQ2, we compare UniLoc with the technique considering only reverting of failure-causing commits.

- **RQ3** How sensitive is UniLoc to different parameter settings and strategies applied? To understand how UniLoc works with different configurations, we changed the parameter values and also created variant approaches by disabling a strategy at a time.

- **RQ4** How effective is our approach for failures to be repaired in source code only, build script only, and both? To answer RQ4, we tried to measure the effectiveness of UniLoc on specific types of failures.

- **RQ5** How effective our approach is for different type of CI failures? To answer RQ5, we tried to measure the effectiveness of UniLoc on test failures and other build failures.

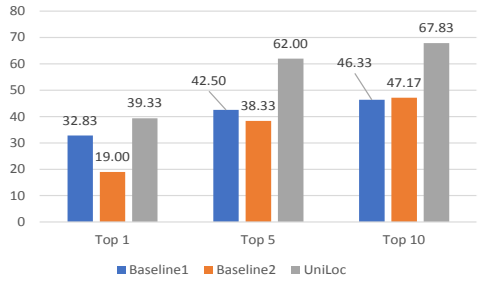


Figure 6.4: Top-N Comparison between Baselines and UniLoc

6.5.4 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 a state-of-the-art IR-based FL technique BLUiR [177](**Baseline2**) 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 [177]—a state-of-the-art source-code-oriented IR-based FL technique according to recent studies [75, 122]. Since 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. 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. Note that the identification of $PART'_f$ is based on our technique described in Section 6.4.1, so both our baselines already partly take advantage of our query optimization technique (directly using the whole build log as query will result in much worse results similar to random file selection due to noises in the log).

Figure 6.4 and Figure 6.5 present the effectiveness comparison between baselines and Uni-Loc. According to Figure 6.4, 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%. In comparison, UniLoc's Top-1, Top-5, and Top-10 values are 39.33%, 62%, and

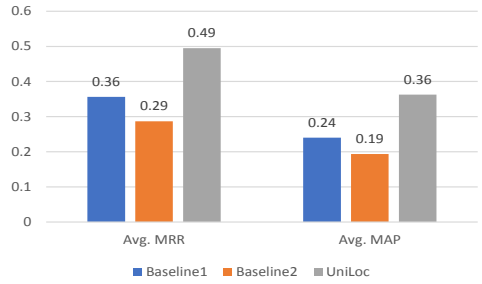


Figure 6.5: MRR and MAP Comparisons between Baselines and UniLoc

67.83%. UniLoc largely outperformed Baseline1 and Baseline2 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 and UniLoc. In Figure 6.5, the Baseline1 technique obtained 0.36 MRR and 0.24 MAP and the Baseline2 obtained 0.29 MRR and 0.19 MAP. While UniLoc achieved 0.49 and 0.36 as MRR and MAP respectively. UniLoc worked more effectively than the baselines.

Specifically in Figure 6.5, UniLoc has wider value ranges of both MRR and MAP than baselines. It means that UniLoc's effectiveness varied a lot among different CI failures.

Finding 1: *UniLoc outperformed both Baseline1 and Baseline2 for all metric, specifically significant higher Top 10 value indicates that UniLoc works better for different type of build 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 purely 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 and MAP metrics. We also compared Change History Based approach with UniLoc. Table 6.5 shows the effectiveness comparison in between the change based approach and our

Table 6.5: Effectiveness Comparison between Change History Based Approach and UniLoc

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

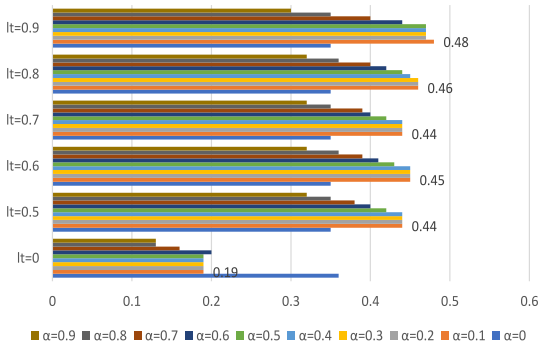


Figure 6.6: MRR for Different Parameter Value on Tuning Dataset

proposed approach. The change based approach achieves 0.38 MRR and 0.28 MAP (compared to 0.49 MRR and 0.36 MAP 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 6.6, 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 $lt = 0.9$, $\alpha = 0.1$.*

There are three strategies applied in UniLoc: S1—query optimization (Section 6.4.1), S2—search space optimization (Section 6.4.2), and S3—file ranking optimization (Section 6.4.4). To understand how different strategies influence UniLoc's effectiveness, we also created three variants

Table 6.6: Effectiveness Comparison between variant approaches, Baselines, and UniLoc

Approach Name	Top 1	Top 5	Top 10	MRR	MAP
Baseline1	31.33%	39.00%	42.83%	0.36	0.24
Baseline2	19.00%	38.33%	47.17%	0.29	0.19
V1—Without Query Optimization	33.50%	59.50%	66.16%	0.44	0.33
V2—Without Search Space Optimization	13.16%	27.00%	35.00%	0.21	0.14
V3—Without File Ranking Optimization	16.67%	36.50%	42.83%	0.26	0.19
UniLoc	39.33%	62.00%	67.83%	0.49	0.36

of our tool: V1—a variant without applying S1, V2—a variant without S2, and V3—a variant without S3.

Table 6.6 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.*

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 6.7, 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 6.7, 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

Table 6.7: Effectiveness Evaluation for Different Fix Types

Fix Type	Approach Name	Fail. Cnt.	Top 1	Top 5	Top 10	MRR	MAP
Source Only	**Baseline1**	430	**30.47**	38.84	42.09	0.35	0.25
	Baseline2		2.56	19.53	29.07	0.11	0.10
	UniLoc		30.00	**53.26**	**59.30**	**0.41**	**0.31**
Build Only	**Baseline1**	78	26.92	34.62	35.90	0.31	0.29
	Baseline2		50.00	**82.05**	**89.74**	0.64	0.60
	UniLoc		**65.38**	**82.05**	87.18	**0.74**	**0.67**
Both	**Baseline1**	92	48.91	66.30	75.00	0.57	0.28
	Baseline2		**69.57**	**89.13**	**95.65**	**0.80**	0.30
	UniLoc		60.87	85.87	91.30	0.71	**0.34**

fewer build script files than source files. Furthermore, two baselines perform very differently in different types of fixes, but UniLoc is more balanced and always has best or close-to-best performance. Since it is not possible to know the type of fixes in advance, UniLoc's balance will help it achieve 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 violate our expectation. After detailed investigation, we found that, even after our query optimization, the build logs still contain too much noises which look similar to AST elements so they misled BLUiR. This result is consistent with UniLoc without optimizations $V2$ or $V3$, as shown in Table 6.6. 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 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). In contrast, **Baseline1** performs better on source-only fixes. Our further investigation (presented in Table 6.8) shows that its high performance mainly comes from source-only fixes of build failures, which are mainly compilation errors and code-style errors. Since file names are often provided in such build failures, it is no wonder that **Baseline1** performed very well on them.

Finding 5: *Baselines' performance varies a lot for different types of fixes, but UniLoc has more balanced performance (always best or close-to-best) among all three types of fixes.*

Table 6.8: Effectiveness for Different Failure Types

Failure Type	Approach Name	Total Count	Only Source Change			Only Build Script Change			Both Change			Aveage	
			Count	MRR	MAP	Count	MRR	MAP	Count	MRR	MAP	MRR	MAP
Test Failure	Baseline1	274	222	0.19	0.14	21	0.06	0.06	31	0.23	0.15	0.18	0.13
	Baseline2			0.09	0.08		0.58	0.51		**0.69**	0.34	0.19	0.14
	UniLoc			**0.35**	**0.27**		**0.66**	**0.58**		0.67	**0.40**	**0.41**	**0.31**
Other Failure	Baseline1	326	208	0.53	0.38	57	0.40	0.37	61	0.75	**0.35**	0.55	0.37
	Baseline2			0.14	0.12		0.66	0.64		**0.85**	0.27	0.36	0.24
	UniLoc			0.47	0.35		**0.76**	**0.71**		0.72	0.31	**0.57**	**0.40**

RQ5: Effectiveness on Different Types of CI Failures. CI failures can be triggered by test failures or build failures. We also clustered UniLoc's evaluation results among these two kinds of failures and present the results in Table 6.8. In Table 6.8, we can observe that all approaches perform better on build failures than test failures, which is reasonable as the latter can be more complicated and involve more files. We also observe the same trends that the two baseline approaches' performance differs greatly in different types of failures and fixes, but UniLoc's performance is more balanced. Since it is possible to know the type of a failure before the FL process, developers can pick the approach that performs better for a specific type of failure. The results show that UniLoc performs better than baselines for both types of failures.

Finding 7: *UniLoc works more effectively on build failures, and performs better than both baselines on both test failures and build failures.*

6.6 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 [119, 169, 200] 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 are fully precise for our evaluation) and 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 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.

6.7 Discussions

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 [7] provides multi-module build with dependency information with the help of Ivy [20]. In Maven [25], 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 ca-

ses, code instrumentation on one file type may miss root causes of failures, while a comprehensive code instrumentation support can be difficult to implement.

6.8 Related Works

6.8.1 Automatic Bug Localization

Automatic bug localization has been an active research area over the decades [125] [236]. Automatic bug localization techniques can be generally divided into two categories: i) dynamic approaches and ii) static approaches. Dynamic fault localization [159] 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) [45] is the most prominent technique. SBFL usually depends on suspicion score based on program elements executed by the test cases. Tarantula [108] is the early research work on SBFL and subsequent researchers are working to improve the accuracy of the localization technique. Ochiai [44] uses different similarity co-efficient to find more accurate fault localization. Xuan and Monperrius proposed Multric [221], which combines learning-to-rank and fault localization techniques for more accurate localization. Recent work on fault localization: Savant [54] uses likely invariant with learning-to-rank algorithm for fault localization. Since software building process lacks test cases and takes time to build software, dynamic approaches might be overhead for CI environment.

Static fault localization techniques do not require test cases and execution information. Static fault localization depends on static source code analysis [233] [78] or information retrieval based approaches [200] [236]. Lint [107] is one of the first tool to find fault in `C` programs. FindBug [53] and PMD [26] are prominent static code analyzer for Java source code. Lots of IR-based approaches [236] [177] have been proposed by the researchers for fault localization. BugLocator [236] performs bug localization based on revised VSM model. Saha et al. [177] proposed BLUiR considering source code structure for IR-based fault localization. Recent work on fault localization Locus [211] 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.

6.8.2 Fault Localization Supporting Automatic Program Repair

Over the last few years Automatic Program Repair [112] [89] is gaining popularity. GenProg [89] uses Genetic Programming(GP) for automatic patch generation. RSRepair [164] performs random searching for generating a path. To reduce searching from existing code, PAR [112] 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 [128] uses a probabilistic model to generate a new patch. Van Tonder and Le Goues [197] applied separation logic for automatic program generation. While Wen et al. [212] used AST context information for better program repair. Automatic program repair techniques are also getting popular for automatic build repair. Recently Macho et al. [132] proposed BUILDMEDIC to repair Maven dependency failure. HireBuild [99] 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 6.8.1, 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.

6.8.3 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 [51] proposed a Petri-net based model. Adams et al. [46] 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. [213] proposed BLIMP for build change impact analysis generated from the build dependency graph. Xia et al. [219] proposed a machine learning based model to predict build co-changes. While from source code change history, Macho et al. [131] proposed model to predict build configuration changes. McIntosh et al. [142] performed a large study to find relation in between build maintenance and build technology. SYMake [188] uses a symbolic-evaluation based technique to detect common errors in Make files. On the study of building errors, Hassan et al. [94] 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. [48]. 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. [180] proposed an approach to identify bad smells in configuration files.

6.9 Conclusion and Future Work

Most existing approaches(e.g., Locus [211], BRTracer [216], BLUiR [177]) 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. 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 sus-

picion 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 and MAP as 0.36, which outperforms baseline approaches for all type 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.

CHAPTER 7: RUDSEA: RECOMMENDING UPDATES OF DOCKERFILES VIA SOFTWARE ENVIRONMENT ANALYSIS

In this chapter, we discussed RUDSEA, a novel approach to recommend updates of dockerfiles to developers based on analyzing changes on software environment assumptions and their impacts. Significant portion of this work has been presented at the following venue [95]:

- F. Hassan, R. Rodriguez and X. Wang, "RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis," 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 2018, pp. 796-801.

7.1 Introduction

Modern software often depends on a large variety of environment dependencies to be properly deployed and operated on production machines. Databases, application servers, system tools, third-party libraries, and supporting files often need to be well installed and configured before software execution, and thus may cause tremendous effort and high risks during software deployment. This is not one-time but continuous cost due to the fast software evolution and delivery nowadays.

A practical approach to alleviate this effort is to use container images. A container image is a stand-alone and executable package of a piece of software with all its environment dependencies, including code, runtime, system tools, libraries, file structures, settings, etc. It can be easily ported and deployed to other machines, but is much lighter-weight than traditional virtual machines which can achieve similar goals.

Despite the large benefit brought by container images during software deployment, they also increase the effort of software developers because they need to generate and maintain the image configuration files which describe how the container images can be constructed with all environment dependencies, such as what tools and libraries should be installed and how the file structure should be set up. A recently study [72] on Dockerfiles by Cito et al. shows that in top projects a docker file is averagely revised 5.8 times each year (note that there can be multiple dockerfiles

in one project, and the average and maximum number of dockerfiles per project in our dataset is 4.9 and 41). Such a task can be tedious and error prone because (1) modern software typically relies on many environment dependencies, and due to fast evolution of software requirements and underlying frameworks, such dependencies also need to be changed very frequently; (2) some environment changes (e.g., automatic system updates, environment changes during installation of irrelevant software) can happen without any developer actions so developers may even not be aware about them; (3) developers can easily neglect environment dependencies of their software when they set up or change them because the changes are made in the operating system instead of the software itself; and (4) many environment dependencies (e.g., system tools, supporting files) cannot be checked during software compilation but only used at runtime, so they can be easily missed during compilation and testing (which is hardly thorough). Once an incomplete or erroneous image configuration file is being used, the container image will also be incomplete or contains errors, which may cause failures in production machines.

In this paper, we propose a novel technique, RUDSEA, to help developer update container image configuration files more easily and with more confidence on their correctness. Specifically, based on an existing image container file, RUDSEA first tracks the accesses to the system environment from software source code and build configuration files. Such accesses are extracted as environment-related code scope. Then, for each code commit, RUDSEA traces its impact on environment-related code scope and automatically determines whether certain items in the image configuration file should be updated accordingly. Based on the type of code impact and configuration items, RUDSEA further recommends the actual updates that should be made on the items. We implement our technique for Docker[1], which is currently the dominating framework in container images for both software industry and open source community, and the image configuration files for docker are called *dockerfiles*. Notef that, although the implementation and evaluation of this research focus on docker images and dockerfiles, the general approach is applicable to other container images such as Kubernetes, OCI, etc.

[1]https://www.docker.com/

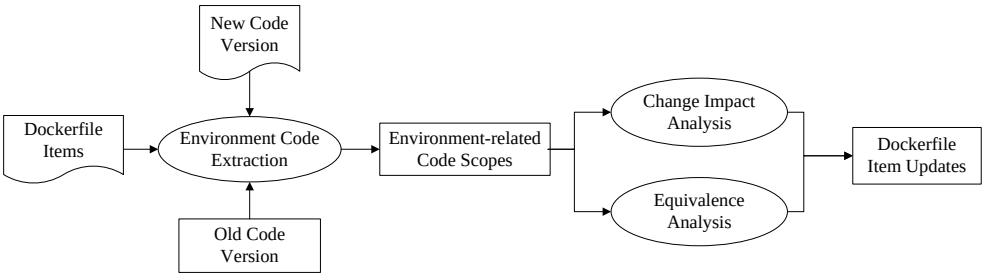


Figure 7.1: RUDSEA Overview

To evaluate RUDSEA, we carried out an experiment on a dataset of 375 dockerfiles in 40 software projects collected from GitHub. Our evaluation shows that RUDSEA correctly recommends update locations for 941 of 1,199 instruction updates in dockerfiles, with a precision of 49.8%. Furthermore, RUDSEA is able to correctly recommend the actual revision for 529 of the 1,199 dockerfile updates. To sum up, this paper makes the following contributions.

- RUDSEA, a novel technique on automatically recommending update locations and contents for dockerfiles during software evolution.
- A dataset of dockerfiles and their corresponding historical versions as benchmarks for future research on this topic.
- An empirical evaluation of RUDSEA's effectiveness on real world dockerfiles.

The rest of this paper is organized as follows. First, we will introduce some background knowledge about dockerfiles in Section 2. Then, we describe our approach and detailed techniques in Section 3. After we present our evaluation results in Section 4, we discuss some related works in Section 5 and concludes in Section 6.

7.2 Background

In this section, we will introduce some background knowledge about dockerfiles. A dockerfile typically consists of three parts. The first part (*From*) specifies an existing container image that the configured image is based upon. Some examples of existing images may include a clean Ubuntu

system of a certain version, or a publicly available image prepared with Java, Android SDK and databases. The second part (*Parser Directives*) describes rules such as escape characters on parsing the rest of the dockerfile, and is optional. The third part (*Environment Replacements*) is the main part of the dockerfile, and describes how the image should be constructed with a sequence of instructions. The major types of instructions are listed below.

- *RUN & WORKDIR*: executing a system command or executable within the working directory specified by *WORKDIR*.
- *CMD & ENTRYPOINT*: setting the default command (*CMD*) to be executed and arguments(*ENTRYPOINT*) to be use when executing the container image.
- *LABEL*: Setting environment variables in the container image.
- *EXPOSE*: exposing a network port in the container image.
- *ENV*: defining a variable to be used in the rest of the dockerfile.
- *ADD / COPY*: add a new directory / file in the file system of the container image, and copy directories / files from hosting system to the image.

From the list, we can see that three types of instructions will be updated frequently during software evolution, which are *RUN* instructions (updating versions of tools / libraries to be installed), *Label* instructions (updating environment variables), and *Add / COPY* instructions (changing default file structures). By contrast, other instructions are either typically stable (e.g., *EXPOSE*, *CMD & ENTRYPOINT*) or used only in the dockerfile itself (e.g., *ENV*). Therefore, our paper focuses on the updates of *RUN*, *LABEL*, and *ADD / COPY* instructions.

7.3 Approach

As shown in Figure 7.1, our approach consists of two major components. The first component extracts software code that is related to the items in dockerfiles. Here the software code base

includes source files, build configuration files, and property files. The core part of this component is value dependency analysis, and we apply it to both old and new versions to acquire the results for both versions. The second component receives the analysis results of two code versions and generates the actual updates. It leverages change impact analysis to determine whether the code change may affect the environment-related code, and equivalence analysis to check whether new code is added as the equivalent part of known environment-related code.

7.3.1 Extracting Environment-related Code Scope

The major challenge of extracting environment-related code is the complicated interface between software and its environment. While software libraries and their versions are typically listed in build configuration files (e.g., `makefile` for GNU Make, `pom.xml` for Maven, `build.gradle` for Gradle), references to file paths and environment variables are often scattered in source code, build configuration files, property files, etc. A thorough definition of all possible environment interfaces requires huge manual effort, and the definition can easily be out-of-date due to quick evolution of the underlying development frameworks, build configuration tools, and their various plug-ins.

To overcome this challenge, RUDSEA uses a different solution. Our intuition is that, all the environment related code, no matter how they interface with environment, must refer to the values in the items of dockerfiles. Note that here we assume that the original version of the dockerfile is a correct one. Simply put, we can search for the values from dockerfile items in the constant string values in various source files, since such values must be used when software interfaces with the environment.

However, a simple keyword search does not work, because developers frequently use string concatenations and value assignments to generate runtime values from the string constants. For example, the dockerfile may refer to a file path `/home/project-name/foo/bar`, while in the source code, the file path may be a string concatenation expression such as `"/home/" + project + "/" + module + "/bar/"`, where `project` and `module` are variables for flexibility of chan-

ging sub-projects and modules. In such cases, the original values will not be detectable with simple keyword search, but string concatenations and assignments need to be considered. In our initial implementation of RUDSEA, we consider only string concatenations, as we find that other string operations are rarely used in generating library names, file paths, and environment variable values.

Therefore, RUDSEA uses a two-stage approach, which first locates the initial string constants which are long enough substrings of a dockerfile item. Then, RUDSEA performs value dependency analysis to compute additional values through manipulating these initial string constants. As our analysis is light weight, we need only a parser and known string concatenation functions (which are typically only several, and very similar among all programming languages) for each programming language used in the software project.

Locating Initial String Constants

The first step of locating initial string constants is to extract dockerfile item values from dockerfiles. To achieve this, we use a dockerfile parser to extract all argument values of *RUN*, *Label*, and *Add / COPY* instructions. Since *RUN* instructions often take Linux utility commands (e.g., *mkdir*, *apk-get install*) as their parameters, and such commands are not necessarily referred in the software code base, we filter out all such commands from dockerfile item values.

After collecting the list of dockerfile item values, RUDSEA extracts all string constants from the software code base, and verifies whether their length is over 3 and is a substring of any dockerfile item values. If so, the string constant is added to the set of initial string constants. In particular, given a string constant str, and a set of dockerfile item values D, Formula 1 presents a boolean function env which checks whether str is an initial string constant. In the rest of the paper, the set of initial string constant is denoted as $Init$.

$$env(str) = len(str) \geq 3 \wedge \exists d \in D, d.contains(str) \tag{7.1}$$

In the formula, we use $len(x)$ to represent the length of string x, and $x.contains(y)$ to represent string y is a substring of x. We will use this check function also in our value dependency analysis

to make the abstract domain bounded. Based on the initial string constants, RUDSEA performs value dependency analysis which checks how string constants are combined with each other to form more values, and tracks the string manipulation process.

Value Dependency Analysis

The value dependency analysis in RUDSEA is a static analysis on string concatenations and assignments within the software code base. Value dependency analysis uses an abstract domain $< \Gamma, T >$. Γ is a set of mappings from the set of string variables V in the software code base, to sets of string values generated from the set of string constants (S) in the code base. Specifically, Γ is defined in formula 2.

$$\Gamma = \{var \rightarrow L \mid var \in V \wedge L \subset S^*\} \tag{7.2}$$

For each value in L, we also track the locations of string constants that form each value in T, so basically T is a mapping from a string value in L to a set of program points.

Why RUDSEA does not use automatons to represent string values? In our value dependency analysis, to track string concatenations and assignments, we use a string set domain instead of an automaton as in string taint analysis [208] for two reasons as follows. First, string taint analysis (and also the original string analysis [69]) uses the Mohri-Nederhof algorithm to handle strongly connected components in string dependencies, and generates an approximate automaton, which is a slow process and typically results in over-approximation and affect analysis accuracy. Second, in string taint analysis, the tracing from original string constants to the final string values is at character level, which makes it difficult to propagate updates from original string constants to the final string values.

Despite the accuracy, efficiency, and straightforward tracing provided by the string value set domain, its major drawback (and why it cannot be used in general string analysis) is that it is not bounded. When a string variable is written within an unbounded loop or recursive method, the possible values of the variable can be infinite.

In the specific application scenario of RUDSEA, we find that this problem can be solved. Our idea is to use the env function to bound the string value set in our domain. The intuition is that, if a possible value of a string variable does not satisfy env function, it will not be a reference to dockerfile item values, and thus can be discarded. Therefore, given that dockerfile item values are finite, all string values in our abstract domain will be perfectly bounded (without any accuracy loss regarding reference to dockerfiles) by the dockerfile item values through env function. In particular, the transfer functions of value dependency analysis on string initialization, string assignments, and string concatenations are defined in

Once value dependency analysis converges at a fixed point, we can tell for each variable, what are its possible values (satisfying env functions) and the original string constants and string concatenations used in forming each value. If a string variable var contains a value val that is identical with any dockerfile item value, we will consider var and all the statements used in forming val as in the environment-related code scope. Specifically, we denote all the dockerfile item values generated from software code base with value dependency analysis as Gen, and Gen is formally defined in Formula 3. Recall that D is the set of all dockerfile item values extracted from the dockerfiles.

$$Gen = \bigcup_{var \in V} \Gamma(var) \cap D \tag{7.3}$$

Then the environment-related code scope can be formally defined as in Formula 4. Recall that T is a part of our abstraction domain which maps any string value in Γ to program points involved in generating the value. Gen and T will be further used in our Dockerfile change generation stage.

$$Scope = \bigcup_{val \in Gen} T(val) \tag{7.4}$$

7.3.2 Dockerfile Change Generation

Given a new software version, RUDSEA's dockerfile change analysis tries to find out what updates on the code will affect items in dockerfiles. Note that RUDSEA does not take single code

commit as its input, because dockerfiles are often not updated until a new release so there may be many code commits in between. Environment change analysis include the change impact analysis which examines whether known environment-related code scope will be affected by the changes, and the equivalence analysis which examines whether a new environment-related code scope is added.

Change Impact Analysis

In the change impact analysis, RUDSEA will perform value dependency analysis on the new version of the software, and map the analysis results (string constants and statements involving string concatenations / assignments) with that from the original version with a file difference tool. In the rest of this section, we denote Gen, T, and $Scope$ generated from the value dependency analysis on the original version as Gen_{old}, T_{old}, and $Scope_{old}$, while the corresponding results on the new version as Gen_{new}, T_{new}, and $Scope_{new}$. We further define the set of variables that has at least one possible value in Gen as $Gvar$. We refer to such variables as *docker variables*. Similarly, we have $Gvar_{old}$ and $Gvar_{new}$. Note that $Gvar$ is formally defined in Formula 5.

$$Gvar = \{var \mid var \in V \wedge \Gamma(var) \cap Gen \neq \emptyset\} \tag{7.5}$$

The intuitive assumption behind our change impact analysis is as follows. If a variable var has a dockerfile item value in its possible value set $\Gamma(var)$ (i.e., var is a docker variable), it is likely to be used for environment interfacing. Therefore, if it holds a different set of values in the new version, the new set of values are likely to be also used for environment interfacing and should be added to the dockerfile. Furthermore, if a docker variable is deleted in the new version, the corresponding dockerfile item value may also need to be deleted if no other docker variables hold the same value in the new version.

As an example, consider a variable var having a possible value `"/home/foo/bar"` in the old version, and the value is a dockerfile item value. In the new version, if the same variable has a possible value `"/home/foo/bar2"`, then it is likely that we should add `"/home/foo/bar2"`

to dockerfiles. In particular, if "`/home/foo/bar`" is no longer in $\Gamma_{new}(var)$, we should replace "`/home/foo/bar`" with "`/home/foo/bar2`". If "`/home/foo/bar`" is still in $\Gamma_{new}(var)$, we should insert a new instruction that performs exact the same operation on "`/home/foo/bar2`" as on "`/home/foo/bar`". If the variable var is deleted in the new version, and no other docker variables has "`/home/foo/bar`" in its possible values, the value should be deleted from the dockerfile.

A complication in this process is when a old docker variable (var in $Gvar_{old}$) holds multiple values in Gen_{old}, or hold other values that are not in Gen_{old}. In such cases, when the possible values of var contains some new value in the new version, it is hard to tell which old value this new value is replacing or complementing. Our solution is to compare their forming process stored in T. Given a new value $newv$ in $\Gamma_{new}(var)$, we compare $T_{new}(newv)$ with each of the old values $oldv$ in $\Gamma_{old}(var)$, and map this new value to an old value $oldv$ whose forming process $T_{old}(oldv)$ is most similar to $T_{new}(newv)$. Specifically, we measure similarity by the size common program points between $T_{old}(oldv)$ and $T_{new}(newv)$.

Equivalence Analysis

While change impact analysis is able to recommend dockerfile updates related to existing dockerfile item values. There are also other cases where a new environment dependency is added. RUDSEA needs to also detect those cases and find out where the insertions need to be made.

To solve this issue, we develop equivalence analysis which checks which two program points have similar usage in the program. They are considered equivalent program points. In our analysis, we consider similar code inside one basic block or in different alternative blocks (i.e., basic blocks within the same level in a conditional statement). Examples of alternative blocks are `if` and `else` blocks within one conditional statement, or `case` blocks within one switch statement.

The intuition behind equivalence analysis is that if a writing statement to a string variable $equiv$ is inserted as a equivalent program point of a writing statement s which writes to a docker variable var with dockerfile item value val, the inserted writing statement will be considered as

a new docker variable, and its possible values will be recommended for insertion into dockerfiles. For each possible value of $equiv$, RUDSEA recommends an insertion of a new instruction that performs exact the same operation on $equiv$ as on val.

7.3.3 Implementation

We implemented the value dependency analysis of RUDSEA for Java, PHP, and Gradle. To support Maven, simple property files, and XML property files, we further convert all dependencies and property definition in such files as string constant assignments (i.e., assignment of property value to property name, and dependency values to a special variable "dependency"), thus they can be handled by the dockerfile-update generation component of RUDSEA.

7.4 Evaluation

To evaluate the effectiveness of RUDSEA, we carried out an experiment on a set of software projects with dockerfiles, and used their version histories as ground truth to check how accurate RUDSEA's recommendation is. Specifically, we try to answer the following two research questions.

- **RQ1:** How effective is RUDSEA on recommending update locations in Dockerfiles?
- **RQ2:** How effective is RUDSEA on recommending updates in Dockerfiles?
- **RQ3:** What are the major reasons causing RUDSEA to fail on recommending correct updates?

In the rest of this section, we introduce the dataset construction, evaluation metrics, evaluation results, and threats to validity in the following four subsections, respectively.

7.4.1 Dataset of Dockerfiles

We collected a set of Docker-using open source projects in Github[2]. In particular, we searched through top Java and PHP projects by number of stars and check whether the project contains dockerfiles. If so, we added the project into our dataset. We stopped after we collected 20 PHP projects and 20 Java projects. Then, we checked the history of the dockerfiles in these projects. In some projects, dockerfiles have their own repository, so we gathered the dockerfiles from there. In some other projects, dockerfiles are attached with each release (so they do not have a version history), we collected all dockerfiles from all releases so that they form a version history. From the version history of dockerfiles, we used diff to generate ground truth updates of dockerfiles. We further removed all internal updates of dockerfiles (e.g., updates of comments, refactorings). Finally, we acquired a dataset of 375 external updates of dockerfiles, each of which can be ascribed to one or more updates in the source code and / or build configuration files. In our evaluation, we use the updates in the source code and / or build configuration files as input, and the corresponding dockerfile updates as output. It should be noted that each update may involve multiple instruction updates. In total, the dockerfile updates include 1,199 instruction insertions, revisions, and deletions.

One question should be studied is how large the dockerfiles are, so that we can see how difficult the update localization is. To answer this question, we further performed an empirical study on our dataset. In the 40 Java and PHP projects, there are 197 dockerfiles in total. The number of dockerfiles in a single project ranges from 1 to 41, and the average number is 4.9. The number of valid lines (excluding blank and comment lines) in dockerfiles varies from 1 to 64 lines, and the average is 28 lines. Since there are often multiple docker files in one project, the average number of dockerfile lines in a project is 137 lines, and the number of lines ranges from 12 lines to 622 lines. Although dockerfiles are relatively smaller than source code, they are condense formatted (i.e., there are often multiple commands to be executed in one line), and their dependency on the code is latent. So the localization of updates is still a difficult problem.

[2]The dataset is available at https://sites.google.com/site/rudseaproject/

Table 7.1: Effectiveness of RUDSEA on recommending update locations

Project	# of Actual Inst. Updates	P (%)	R (%)	F (%)
PHP	720	53.9	79.7	64.3
Java	479	44.5	76.6	56.3
All	1,199	49.8	78.5	60.9

7.4.2 Metrics

In our experiment, we use the traditional metrics of precision, recall, and F-score to measure the effectiveness of techniques. We consider a recommended location to be correct, if the recommended instruction to be updated is revised, deleted, or have another instruction inserted before of after it in the real version history.

For a recommended update to be correct, we require the recommendation has the same type (insertion, update, or deletion), same instruction type, and argument value. Here we consider equivalent updates as also correct. For example, recommending a same insertion at a different location from the real insertion is also considered correct as long as the location difference does not cause difference in semantics.

7.4.3 Evaluation Results

To answer **RQ1**, we present our evaluation results in Table 7.1. In the table, we present the type of projects, the number of actual instruction updates, precision, recall, and F-score in Columns 1-5, respectively. From the table we can see that RUDSEA is able to achieve high recall (averagely 78.5%) and acceptable precision (averagely 49.8%) in recommending update locations. Note that, since averagely less than four updates are performed in each commit, achieving a precision at around 50% means that developers need to inspect averagely eight locations, and finding four of them correct.

To answer **RQ2**, we present the results in Table 7.2 with the same format. From the table we can see that RUDSEA can achieve an average recall of 44.1% on recommending direct updates. This means that RUDSEA can recommend exactly correct updates for 529 of 1,199 instruction

Table 7.2: Effectiveness of RUDSEA on recommending updates

Project	# of Actual Inst. Updates	P (%)	R (%)	F (%)
PHP	720	28.7	42.6	34.3
Java	479	27.0	46.3	34.1
All	1,199	28.0	44.1	34.3

updates, which may save a large amount of effort of developers. Compared with the recall on location recommendation, we can see that for the updates RUDSEA successfully recommends locations, about 56% (529) are exactly correct updates. To answer **RQ3**, we studied the remaining 412 incorrect updates and find the errors mainly fall into three categories.

First, RUDSEA may insert an instruction at a wrong location. For simplicity, when RUDSEA finds that a docker variable has a new value which can be mapped to a dockerfile item value v in change impact analysis or equivalence analysis, RUDSEA always insert an extra instruction after the instruction handling v. Since instructions in dockerfiles are executed in sequence, such an insertion location may be wrong, especially when v is handled in a long instruction concatenated with "&&". This category accounts for 207 incorrect updates and we believe that most of them can be resolved by more fine-grained rules on dockerfile insertions.

Second, although RUDSEA correctly recommends an insertion, the inserted argument may not be correct. Developers sometimes add extra parameters to the *RUN* instructions they added, but RUDSEA is not able to recommend such parameters as it does not understand their semantics. This category accounts for 90 incorrect updates.

Third, when a docker variable cannot be mapped to a variable in the new version, RUDSEA simply deletes dockerfile item values in its possible value set from dockerfile. Some complicated version updates of the software cause difficulties in finding correct mapping of variables between versions and thus RUDSEA may delete a value that should be revised. This category accounts for 65 incorrect updates and we believe that they can be partly resolved by using more precise version diff tools.

7.4.4 Threats to Validity

The major threat to the internal validity of our evaluation is whether the ground truth updates we used in our experiment are all correct. Although we use real-world updates, developers may make erroneous updates or miss some updates, which may cause inaccuracy in our results. Also, the implementation of RUDSEA may be not perfect and involve bugs. The major threat to the external validity is that our evaluation results apply to only the subject projects and updates, or only Java / PHP projects. To reduce this threat, we use projects from Github based on different programming languages.

7.5 Related Work

Studies and Analyses of Dockerfiles. With the increase of software complexity and components, managing of software dependencies [147] and test dependencies [148] has become an important problem. Tufano et al. [195] studied on broken snapshots and likely causes behind broken snapshots. Recent research work on scientific artifact reproduction [64] discussed about the uses of Docker to address the challenge of operating system virtualization, cross-platform portability, and reusable software components. Cito et al. [70] discussed about the rise of Docker adoption in industry, and performed an empirical study on dockerfiles [72]. Rahman and Williams [168] performed an empirical study on the type of defects in dockerfiles. Docker is also used for lightweight virtualization for developers for distributed application development, build and ship [110].

Analysis of Building Configuration Files. As build configuration files are getting complex and diverse, research on build configuration file is getting importance that includes dependency analysis, migration of build systems and empirical studies. To keep consistency during revision, Adams et al. [46] proposed a framework to generate dependency graph of build configuration files. Al-Kofahi et al. [49] proposed a fault localization technique for make files, and SYMake [188] uses a symbolic-evaluation-based technique to detect common errors in Makefile. Following works by Zhou et al. [237] and Al-Kofahi et al. [50] try to find configuration values exercising different parts of makefiles. Shambaugh [179] developed a verifier for puppet configuration script, and Sharma

et al. [180] proposed techniques to detect bad smells in configuration files. Recently, Hassan et al. studied the reproduction of building environments [94, 97], and performed AST level analysis to generate fix patch for build configuration files [99] .

String analysis. String analysis [69] is a static analysis technique to estimate possible values of string variables. String analysis has been applied to detecting vulnerabilities [208, 225], repair web interfaces [207], software internationalization [204], inter-component communication analysis [155], etc.

7.6 Conclusion and Future Work

In this paper, we present RUDSEA, which is a novel approach to recommend updates for dockerfiles during software evolution. RUDSEA leverages tracks environment accesses from code to extract environment-related scopes from the old software version and the new software version. Then, RUDSEA generates updates from the two versions of analysis results. Our evaluation on 40 projects and 1,199 real-world instruction updates shows that RUDSEA can recommend correct update locations for 78.5% of the updates, and correct updates for 44.1% of the updates, with moderate false positives.

CHAPTER 8: CONCLUSIONS

Our study on software build-ability comes up with the first study on the build failures found in top Java projects, and whether such failures can be resolved with automatic tools. We have constructed a taxonomy for the root causes of build failures and study the distribution of build failures in different categories. Specifically, we found 91 build failures in 86 of the 187 Java projects that use Maven, Ant, and Gradle for their building process, and 52 of the build failures can be resolved automatically, and additional six build failures have the potential to be resolved automatically.

For stall avoidance in CI environment, our proposed approach can predict build outcome with over 87 percent F-Measure for all build systems in CI environment. This build prediction model will help developers to get early build outcomes without making the actual build. This model even can also be helpful for reducing CI computation resources.

According to the TravisTorrent dataset of open-source software continuous integration, 22% of code commits include changes in build script files to maintain build scripts or to resolve build failures. Automated program repair techniques have great potential to reduce the cost of resolving software failures, but the existing techniques mostly focus on repairing source code so that they cannot directly help resolving software build failures. Our proposed work HireBuild is the first approach for automatic build fix candidate patch generation for Gradle build script. Our solution works on automatic build fix template generation based on build failure log similarity and historical build script fixes. Our experiment shows that our approach can fix 11 of 24 reproducible build failures, or 45% of the reproducible build failures, within a comparable time of manual fixes.

Although our HireBuild work can repair build scripts in CI environment, a special challenge in continuous integration is that its failures are often a combination of test failures and build failures, and sometimes both source code and various build scripts need to be touched in the repair of one failure. However, most existing fault localization and repair techniques target source code and test failures only. A more realistic scenario in practice is that multiple types of failures happen

simultaneously and can be ascribed to multiple types of files. 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 and build script.

CHAPTER 9: FUTURE DIRECTIONS

9.1 System Event-Driven Fault Localization of Test and Build Failure in CI Environment

Automatic bug localization has been an active research area over the decades [125] [236]. Automatic bug localization techniques can be broadly divided into two categories: i) dynamic approaches and ii) static approaches. Our current implement fault localization technique is static in nature. Dynamic fault localization [159] requires the execution of program and test cases to identify precise fault location. Dynamic fault localization techniques need preprocessing of the code or underlying platform, as well as a precise reproduction of the failure. Among the dynamic fault localization techniques, spectrum-based fault localization(SBFL) [45] is the most prominent technique. SBFL usually depends on the suspicion score based on program elements executed by the test cases. Although traditional SBFL dependents on test cases for suspicion score, for CI failures, there might be any involved test cases. With our proposed approach UniLoc(See Section 6), we tired fault localization of source code and build script using IR-based technique. But to scale approach for heterogeneous errors involving different file types, we need to have a unified trace environment to localize faults in a more precise manner. To mitigate the issue, we planned to utilize system trace information such as Ptrace, Strace, etc. to our earlier developed FL research work. The proposed work can have the advantage over traditional FL technique in three folds: i) mitigate instrumentation effort, ii) provide pinpoint error generation path, and iii) can work software solutions developed in multi-programming language. This enhanced FL technique with execution context information can also be helpful for program repair technique enhancement.

9.2 Repair of Build Script and Source Code in Combination CI Failures

Various fault localization and code repair techniques are designed to help developers perform code repair more easily and timely. However, a special challenge in continuous integration is that

its failures are often a combination of test failures and build failures, and sometimes both source code and various build scripts need to be touched in the repair of one failure. Our recent study on 1,187 CI failures on TravisTorrent Dataset [62] also confirms that 10.8% of CI-failure repair commits contain only build script revisions, and 25.6% CI-failure repair commits involve build script revisions. Furthermore, although it is possible to differentiate test failures and build failures, it is often not possible to ascribe the failure to source code or build scripts just based on the failure type. Our study (see Section 6.5.1) shows that build failures may be repaired by source code revisions, and test failures may be repaired to build script revisions. Our implemented HireBuild focused only build script failures. HireBuild cannot fix CI failures that require fix in source code or both source code and build script fix together. Moreover, we plan to increase training and testing data size for better coverage of build failures with better evaluation and perform a study on patch quality for the patches generated by our tool. Apart from that, we are planning to apply search-based techniques such as genetic programming with fitness function on our patch list to better rank our generated patches and apply a combination of patches.

LIST OF PUBLICATIONS

[1] F. Hassan and X. Wang, "HireBuild: an automatic approach to history-driven repair of build scripts," In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). ACM, pages 1078-1089.

[2] F. Hassan, S. Mostafa, E. S. L. Lam and X. Wang, "Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 38-47.

[3] F. Hassan and X. Wang, "Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 157-162.

[4] F. Hassan, R. Rodriguez, and X. Wang, "RUDSEA: recommending updates of Dockerfiles via software environment analysis," In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, pp 796-801.

[5] F. Hassan and Xiaoyin Wang, "Mining Readme Files to Support Automatic Building of Java Projects in Software Repositories," 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, 2017, pp. 277-279.

[6] Foyzul Hassan, Chetan Bansal, Nachiappan Nagappan, Thomas Zimmermann, Ahmed Hassan Awadallah, "Exceptions in the Wild:A Large-Scale Study using Search Logs," in preparation for the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2020).

[7] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, Lingming Zhang, "Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis," Accepted in 42nd International Conference on Software Engineering (ICSE 2020).

[8] Foyzul Hassan, Na Meng, Xiaoyin Wang, “UniLoc: Unified Fault Localization of Continuous Integration Failures,” in submission at IEEE Transactions on Software Engineering (TSE).

[9] Foyzul Hassan, “Tackling Build Failures in Continuous Integration”, In Proceedings of the International Conference on Automated Software Engineering (ASE 2019), Doctoral Symposium.

BIBLIOGRAPHY

[1] adcash. https://www.adcash.com/.

[2] andFHEM. https://github.com/klassm/andFHEM.

[3] Android checkout library. https://github.com/serso/android-checkout.

[4] Android lyrics plugin. https://github.com/musixmatch/android-lyrics-plugin-sdk.

[5] Android permission system. https://developer.android.com/guide/topics/permissions/index.html.

[6] Android runtime permissions. https://developer.android.com/training/permissions/requesting.html.

[7] Ant. http://ant.apache.org/. Accessed: 2014-08-30.

[8] Antenna pod. https://github.com/AntennaPod/AntennaPod.

[9] Docker. https://www.docker.com/. Accessed: 2018-12-24.

[10] Flurry. https://y.flurry.com/.

[11] Gerrit code review - a quick introduction. https://review.openstack.org/Documentation/intro-quick.html/. Accessed: 2016-10-20.

[12] Gnu autoconf - creating automatic configuration scripts. http://www.gnu.org/software/autoconf/manual/index.html. Accessed: 2015-10-25.

[13] Google admob. https://www.google.com/admob/.

[14] Google analytics. https://analytics.google.com/.

[15] Google code - project hosting. https://code.google.com/hosting. Accessed: 2009-08-06.

[16] Google play store. https://play.google.com/store.

[17] Gradle task type. https://docs.gradle.org/3.3/dsl/index.html#N1000C/. Accessed: 2017-07-04.

[18] The gradle wrapper. https://docs.gradle.org/current/userguide/gradle_wrapper.html/. Accessed: 2016-10-20.

[19] Ibm. the t. j. watson libraries for analysis (wala). http://wala.sourceforge.net. Accessed: 2012-03-20.

[20] Ivy. http://ant.apache.org/ivy/. Accessed: 2018-08-18.

[21] Java2s jar archive. http://www.java2s.com/Code/Jar/CatalogJar.htm. Accessed: 2016-09-20.

[22] Jgit: Java library implementing the git version control system. https://eclipse.org/jgit/. Accessed: 2017-04-30.

[23] Kubernetes. https://kubernetes.io/. Accessed: 2019-03-25.

[24] Lucene. http://lucene.apache.org/. Accessed: 2018-08-18.

[25] Maven. https://maven.apache.org/. Accessed: 2018-08-18.

[26] Pmd:an extensible cross-language static code analyzer. https://pmd.github.io/. Accessed: 2018-08-18.

[27] Privacy enforcement of protection. https://oag.ca.gov/privacy.

[28] Privacy policy of lost it! http://www.loseit.com/privacy/.

[29] Readme. https://en.wikipedia.org/wiki/README. Accessed: 2016-09-20.

[30] The real falcon prestissimo. https://github.com/TheRealFalcon/Prestissimo.

[31] The sourceforge story. http://web.archive.org/web/20110716044546/http://itmanagement.earthweb.com/cnews/article.php/3705731. Accessed: 2012-04-12.

[32] Stackoverflow. http://stackoverflow.com/. Accessed: 2016-09-25.

[33] Travis-ci. https://travis-ci.org/. Accessed: 2018-12-24.

[34] Uileak: Detecting privacy-policy violations on user input data for android applications, technical report, utsa,. https://xywang.100871.net/uistudy.pdf.

[35] Wiki for privacy policy. https://en.wikipedia.org/wiki/Privacy_policy.

[36] Yelp. https://www.yelp.com/.

[37] The gradle build language. https://docs.gradle.org/current/userguide/writing_build_scripts.html, 2017. Accessed: 2017-04-30.

[38] Gradle build script structure. https://docs.gradle.org/3.5/dsl/, 2017. Accessed: 2017-04-30.

[39] Android Software Development Kit, 2018. https://developer.android.com/.

[40] Java Development Kit, 2018. http://www.oracle.com/technetwork/java/javase/downloads/index.html/.

[41] The Apache Software Foundation, 2018. http://www.apache.org/.

[42] An Introduction to CI/CD Best Practices. https://www.digitalocean.com/community/tutorials/an-introduction-to-ci-cd-best-practices, 2019.

[43] Why Continuous Integration Doesn't Work. https://devops.com/continuous-integration-doesnt-work/, 2019.

[44] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, Dec 2006.

[45] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[46] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*, pages 114–123, Oct 2007.

[47] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the linux build system. *ECEASST*, 8, 2007.

[48] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. Fault localization for build code errors in makefiles. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 600–601, New York, NY, USA, 2014. ACM.

[49] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. Fault localization for make-based build crashes. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 526–530. IEEE, 2014.

[50] Jafar Al-Kofahi, Tien N Nguyen, and Christian Kästner. Escaping autohell: a vision for automated analysis and migration of autotools build systems. In *Release Engineering (RELENG), 4th International Workshop on*, pages 12–15, 2016.

[51] Nasreddine Aoumeur and Gunter Saake. Dynamically evolving concurrent information systems specification and validation: A component-based petri nets proposal. *Data Knowl. Eng.*, 50(2):117–173, August 2004.

[52] Apache. Opennlp, 2010.

[53] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 805–806, New York, NY, USA, 2007. ACM.

[54] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International*

Symposium on Software Testing and Analysis, ISSTA 2016, pages 177–188, New York, NY, USA, 2016. ACM.

[55] Dominic Balasuriya, Nicky Ringland, Joel Nothman, Tara Murphy, and James R Curran. Named entity recognition in wikipedia. In *Proceedings of the 2009 Workshop on The People's Web Meets NLP: Collaboratively Constructed Semantic Resources*, pages 10–18. Association for Computational Linguistics, 2009.

[56] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317. ACM, 2014.

[57] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[58] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 280–289, 2013.

[59] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, pages 1–43, 2014.

[60] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 770–781, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. Technical report, PeerJ Preprints, 2016.

[62] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.

[63] C. Bird and T. Zimmermann. Predicting software build errors, February 20 2014. US Patent App. 13/589,180.

[64] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[65] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[66] Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336. ACM, 1998.

[67] P.W.D. Charles. Project title. https://github.com/charlespwd/project-title, 2013.

[68] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *USENIX Security Symposium*, pages 1037–1052, 2014.

[69] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. Precise analysis of string expressions. In *Static Analysis Symposium*, pages 1–18. Springer, 2003.

[70] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. The making of cloud applications: An empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting on European Software Conference and Foundations of Software Engineering*, pages 393–403, 2015.

[71] Jürgen Cito, Gerald Schermann, Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. *2017 IEEE/ACM*

14th International Conference on Mining Software Repositories (MSR), pages 323–333, 2017.

[72] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. An empirical analysis of the docker container ecosystem on github. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 323–333. IEEE, 2017.

[73] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[74] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *OOPSLA*, pages 313–328, 2008.

[75] Tung Dao, Lingming Zhang, and Na Meng. How does execution information help with information-retrieval based bug localization? In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 241–250, Piscataway, NJ, USA, 2017. IEEE Press.

[76] C. De Roover, R. Lammel, and E. Pek. Multi-dimensional exploration of api usage. In *ICPC*, pages 152–161, 2013.

[77] Fergal Dearle. *Groovy for Domain-Specific Languages*. Packt Publishing, 1st edition, 2010.

[78] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 307–317, New York, NY, USA, 2017. ACM.

[79] John Downs, Beryl Plimmer, and John G. Hosking. Ambient awareness of build status in collocated software teams. In *Proceedings of ICSE*, pages 507–517, 2012.

[80] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[81] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. Association for Computing Machinery.

[82] T. Espinha, A. Zaidman, and H.-G. Gross. Web api growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 84–93, 2014.

[83] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[84] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In Martin Pelikan and Jürgen Branke, editors, *GECCO*, pages 965–972. ACM, 2010.

[85] Jacqui Finlay, Russel Pears, and Andy M. Connor. Data stream mining for predicting software build outcomes using source code metrics. *Inf. Softw. Technol.*, 56(2):183–198, 2014.

[86] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov 2007.

[87] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. Association for Computing Machinery.

[88] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny Van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *ACM SIGPLAN Notices*, volume 49, pages 599–616. ACM, 2014.

[89] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan 2012.

[90] William G. J. Halfond and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *ACM/IEEE Conference on Automated Software Engineering*, pages 174–183, 2005.

[91] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[92] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, June 2012.

[93] Ahmed E Hassan. The road ahead for mining software repositories. In *FoSM*, pages 48–57. IEEE, 2008.

[94] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang. 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)*, pages 38–47, Nov 2017.

[95] F. Hassan, R. Rodriguez, and X. Wang. Rudsea: Recommending updates of dockerfiles via software environment analysis. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 796–801, Sep. 2018.

[96] F. Hassan and X. Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 157–162, 2017.

[97] Foyzul Hassan and Xiaoyin Wang. Mining readme files to support automatic building of java projects in software repositories: Poster. In *ICSE Companion*, pages 277–279, 2017.

[98] Foyzul Hassan and Xiaoyin Wang. An empirical study of continuous integration failures in travistorrent, technical report utsa-cs-tr-002. https://xywang.100871.net/cistudy.pdf, 2018.

[99] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1078–1089, New York, NY, USA, 2018. ACM.

[100] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[101] Djoerd Hiemstra. Using language models for information retrieval. 2001.

[102] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*, ASE 2016, pages 426–437, New York, NY, USA, 2016. ACM.

[103] Jianjun Huang, Xiangyu Zhang, and Lin Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 169–180, 2016.

[104] Z. Huang, S. Wu, S. Jiang, and H. Jin. Fastbuild: Accelerating docker image building for efficient development and deployment of container. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 28–37, 2019.

[105] Hubert Klein Ikkink. *Gradle Dependency Management*. Packt Publishing, 2015.

[106] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *ICSM*, pages 307–316. IEEE Computer Society, 2008.

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

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

[109] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, pages 825–836, 2014.

[110] Muhamad Fitra Kacamarga, Bens Pardamean, and Hari Wijaya. Lightweight virtualization in cloud computing for research. In Rolly Intan, Chi-Hung Chi, Henry N. Palit, and Leo W. Santoso, editors, *Intelligence in the Era of Big Data*, pages 439–445, 2015.

[111] Hannes Kegel and Friedrich Steimann. Systematically refactoring inheritance to delegation in java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 431–440, 2008.

[112] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811, May 2013.

[113] Gary Kumfert and Tom Epperly. Software in the doe: The hidden overhead of"the build". Technical report, Lawrence Livermore National Lab., CA (US), 2002.

[114] Irwin Kwan, Adrian Schroter, and Daniela Damian. Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. *TSE*, 37(3):307–324, 2011.

[115] ShuvenduK. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique RebÃªlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification*, pages 712–717, 2012.

[116] Patrick Lam, Eric Bodden, Laurie Hendren, and Technische UniversitÃ€t Darmstadt. The soot framework for java program analysis: a retrospective.

[117] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *SAC*, pages 1317–1324, 2011.

[118] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–, 2004.

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

[120] X. B. D. Le, D. Lo, and C. L. Goues. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224, March 2016.

[121] Dik L. Lee, Huei Chuang, and Kent Seamons. Document ranking and the vector-space model. *IEEE Softw.*, 14(2):67–75, March 1997.

[122] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. 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*, ISSTA 2018, pages 61–72, New York, NY, USA, 2018. ACM.

[123] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *CC*, pages 153–169. Springer, 2003.

[124] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. String analysis for java and android applications. In *ESEC/FSE*, pages 661–672. ACM, 2015.

[125] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 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*, ESEC/FSE-13, pages 286–295, New York, NY, USA, 2005. ACM.

[126] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *PLAS*, pages 139–160, 2005.

[127] Fan Long and Martin Rinard. Staged program repair in spr. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2015.

[128] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[129] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):13, 2007.

[130] Yoëlle S Maarek, Daniel M Berry, and Gail E Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on software Engineering*, 17(8):800–813, 1991.

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

[132] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically Repairing Dependency-Related Build Breakage. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, page To appear, 2018.

[133] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *ASEJ*, 10(2):183–202, April 2003.

[134] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[135] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[136] Matias Martinez, Thomas Durieux, Jifeng Xuan, Romain Sommerard, and Martin Monperrus. Automatic repair of real bugs: An experience report on the defects4j dataset. *arXiv preprint arXiv:1505.07002*, 2015.

[137] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 287–296, 2003.

[138] Stephen McCamant and MichaelD. Ernst. Early identification of incompatibilities in multi-component upgrades. In *European Conference on Object-Oriented Programming*, pages 440–464, 2004.

[139] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 70–79, 2013.

[140] S. McIntosh, B. Adams, T.H.D. Nguyen, Y. Kamei, and AE. Hassan. An empirical study of build maintenance effort. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 141–150, May 2011.

[141] Shane Mcintosh, Bram Adams, and Ahmed E. Hassan. The evolution of java build systems. *Empirical Softw. Engg.*, 17(4-5):578–608, August 2012.

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

[143] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458, May 2015.

[144] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, May 2016.

[145] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Apache Maven*. Alpha Press, 2010.

[146] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 484–495, 2014.

[147] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: A study on behavioral backward incompatibilities of java software libraries. In *International Symposium on Software Testing and Analysis*, pages 215–225, 2017.

[148] Shaikh Mostafa and Xiaoyin Wang. An empirical study on the usage of mocking frameworks in software testing. In *QSIC*, pages 127–132. IEEE, 2014.

[149] Robert Munro, Daren Ler, and Jon Patrick. Meta-learning orthographic and contextual models for language independent named entity recognition. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 192–195. Association for Computational Linguistics, 2003.

[150] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[151] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[152] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, 2013.

[153] Ansong Ni and Ming Li. Cost-effective build outcome prediction using cascaded classifiers. In *MSR*, MSR '17, pages 455–458, Piscataway, NJ, USA, 2017. IEEE Press.

[154] Jasmina Novakovic. Using information gain attribute evaluation to classify sonar targets. In *17th Telecommunications forum TELFOR*, pages 1351–1354, 2009.

[155] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *International Conference on Software Engineering*, pages 77–88, 2015.

[156] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. *SIGSOFT Softw. Eng. Notes*, 29(6):241–251, October 2004.

[157] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *EMNLP*, pages 79–86. Association for Computational Linguistics, 2002.

[158] Rajesh Parekh and Vasant Honavar. Grammar inference, automata induction, and language acquisition. In *Handbook of Natural Language Processing*, pages 727–764. Marcel Dekker, 2000.

[159] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: Fault localization in concurrent programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

[160] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620, May 2017.

[161] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, 2008.

[162] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of PLDI*, pages 504–515, 2011.

[163] Martin F Porter. Snowball: A language for stemming algorithms, 2001.

[164] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM.

[165] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. ACM.

[166] Cosmin Radoi and Danny Dig. Practical static race detection for java parallel loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 178–190, 2013.

[167] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387, 2012.

[168] Akond Rahman and Laurie Williams. Characterizing defective configuration scripts used for continuous deployment. In *International Conference on Software Testing*, pages 34–45, 2018.

[169] Mohammad Masudur Rahman and Chanchal K. Roy. 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*, ESEC/FSE 2018, pages 621–632, New York, NY, USA, 2018. ACM.

[170] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, 2003.

[171] Adwait Ratnaparkhi. A simple introduction to maximum entropy models for natural language processing. Technical report.

[172] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. 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)*, pages 345–355, May 2017.

[173] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 71–81, New York, NY, USA, 2018. ACM.

[174] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11):701–726, 1998.

[175] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, Aug 1999.

[176] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

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

[178] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at Google). In *ICSE*, pages 724–734, 2014.

[179] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. In *International Conference on Programming Language Design and Implementation*, pages 416–430, 2016.

[180] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 189–200, New York, NY, USA, 2016. ACM.

[181] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security Privacy*, 3(6):41–49, 2005.

[182] Jasmeet Singh and Vishal Gupta. Text stemming: Approaches, applications, and challenges. *ACM Comput. Surv.*, 49(3):45:1–45:46, September 2016.

[183] E. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2015.

[184] Yoonki Song, Xiaoyin Wang, Tao Xie, Lu Zhang, and Hong Mei. Jdf: detecting duplicate bug reports in jazz. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 315–316. ACM, 2010.

[185] Jürgen Starek. A large-scale analysis of Java API usage, 2010.

[186] Michael Steinbach, George Karypis, Vipin Kumar, et al. A comparison of document clustering techniques. In *KDD workshop on text mining*, volume 400, pages 525–526. Boston, 2000.

[187] Matúš Sulír and Jaroslav Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2016, pages 17–25, New York, NY, USA, 2016. ACM.

[188] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Symake: a build code analysis and refactoring tool for makefiles. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 366–369, Sept 2012.

[189] S. H. Tan and A. Roychoudhury. Relifix: Automated repair of software regressions. In *International Conference on Software Engineering*, 2015.

[190] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, pages 83–95, 2015.

[191] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. Automatically generated patches as debugging aids: A human study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74, 2014.

[192] Jesse Tilly and Eric M. Burke. *Ant: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[193] Maksim Tkachenko and Andrey Simanovsky. Named entity recognition: Exploring features. In Jeremy Jancsary, editor, *KONVENS*, pages 118–127. ÖGAI, September 2012. Main track: oral presentations.

[194] Antonio Toral and Rafael Munoz. A proposal to automatically build and maintain gazetteers for named entity recognition by using wikipedia. Technical report, 2006.

[195] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[196] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. Automated decomposition of build targets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 123–133, Piscataway, NJ, USA, 2015. IEEE Press.

[197] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 151–162, New York, NY, USA, 2018. ACM.

[198] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 805–816, New York, NY, USA, 2015. ACM.

[199] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *ICSE*, pages 408–418, 2003.

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

[201] Xiaoyin Wang and Shaikh Mostafa. Autobuilder: Towards automatic building of java projects to support analysis of software repositories. http://xywang.100871.net/TechReport_AutoBuild.pdf, 2014.

[202] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *ISSTA*, pages 199–210. ACM, 2015.

[203] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering*, pages 461–470, 2008.

[204] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Transtrl: An automatic need-to-translate string locator for software internationalization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 555–558. IEEE Computer Society, 2009.

[205] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Locating need-to-translate constant strings in web applications. In *International Symposium on the Foundations of Software Engineering*, pages 87–96, 2010.

[206] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. Locating need-to-externalize constant strings for software internationalization with generalized string-taint analysis. *IEEE Transactions on Software Engineering*, 39(4):516–536, 2013.

[207] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of*

the 2015 10th Joint Meeting on European Softwar Engineering Conference and Foundations of Software Engineering, page 16, 2012.

[208] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *International Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[209] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[210] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, 2009.

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

[212] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1–11, New York, NY, USA, 2018. ACM.

[213] Ruiyin Wen, David Gilbert, Michael G. Roche, and Shane McIntosh. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*, page To appear, 2018.

[214] Peter Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.

[215] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of ICSE*, pages 1–11, 2009.

[216] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 181–190, Sep. 2014.

[217] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 204–214, New York, NY, USA, 2014. ACM.

[218] Wei Wu, B. Adams, Y.-G. Gueheneuc, and G. Antoniol. Acua: Api change and usage auditor. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 89–94, 2014.

[219] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan. Cross-project build co-change prediction. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 311–320, March 2015.

[220] J. Xuan, M. Martinez, F. DeMarco, M. ClÃ©ment, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, Jan 2017.

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

[222] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of

conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[223] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM.

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

[225] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Form. Methods Syst. Des.*, 44(1):44–70, February 2014.

[226] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *MSR*, MSR '15, pages 367–371, Piscataway, NJ, USA, 2015. IEEE Press.

[227] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 253–267. 1999.

[228] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ACM SIGSOFT Software engineering notes*, volume 24, pages 253–267. Springer-Verlag, 1999.

[229] K. Zhai, B. Jiang, and W. K. Chan. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *IEEE Transactions on Services Computing*, 7(1):54–67, Jan 2014.

[230] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *MSR*, pages 182–191. ACM, 2014.

[231] Hongyu Zhang, Hee Beng Kuan Tan, Lu Zhang, Xi Lin, Xiaoyin Wang, Chun Zhang, and Hong Mei. Checking enforcement of integrity constraints in database applications based on code patterns. *JSS*, 2011.

[232] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201. IEEE Press, 2013.

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

[234] Hao Zhong and Na Meng. An empirical study on using hints from past fixes: poster. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 144–145, 2017.

[235] B. Zhou, X. Xia, D. Lo, and X. Wang. Build predictor: More accurate missed dependency prediction in build configuration files. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 53–58, July 2014.

[236] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.

[237] Shurui Zhou, Jafar Al-Kofahi, Tien N Nguyen, Christian Kästner, and Sarah Nadi. Extracting configuration knowledge from build files with symbolic analysis. In *Release Engineering (RELENG), 3rd International Workshop on*, pages 20–23, 2015.

VITA

Foyzul Hassan was born and brought up in Bangladesh, a small, beautiful country in South Asia. He earned his bachelor's degree in Computer Science and Engineering from the Military Institute of Science and Technology(MIST), Bangladesh. Before joining to PhD program, he worked in different roles of software engineering for eight years in Bangladesh. He enrolled in PhD program at The University of Texas at San Antonio (UTSA) in Fall 2015 and started working under the supervision of Dr. Xiaoyin Wang. His main interest is improving software development productivity. Some specific areas of interests are: continuous integration, software build process, script code analysis, program repair, and fault localization, software engineering tools for machine learning (SE4ML), and security vulnerability analysis for development tools. Foyzul actively collaborated with researchers from The University of Texas at Dallas (UTD), Virginia Tech (VTech), University of Colorado Boulder, and Microsoft Research. In Summer 2019, Foyzul worked at Microsoft Research with Research In Software Engineering Group (RiSE) group as a Research Intern. Due to his excellence in research, UTSA computer science department awarded him Outstanding Achievement in Research Award 2019.