Understanding and Generating Patches for Bugs Introduced by Third-party Library Upgrades

by

Yuefei Liu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

© Yuefei Liu 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

During the process of software development, developers rely heavily on third-party libraries to enable functionalities and features in their projects. However, developers are faced with challenges of managing dependency messes when a project evolves. One of the most challenging problems is to handle issues caused by dependency upgrades.

To better understand the issues caused by Third-party Library Upgrades (TLU), in this thesis, we conduct a comprehensive study of the bugs caused by dependency upgrades. The study is conducted on a collection of 8,952 open-source Java projects from GitHub and 304 Java projects on Apache Software Foundation (ASF) JIRA systems. We collect 83 bugs caused by inappropriate TLUs in total. Our inspection shows that TLUs are conducted out of different reasons. The most popular reason is that the project is preparing for release and wants to keep its dependencies up-to-date (62.3%). Another popular reason is that the older version of a dependency is not compatible with other dependencies (15.3%). Our inspection also indicates that the problems introduced by inappropriate dependency upgrades can be categorized into different types, i.e., program failures that are detectable statically and dynamically. Then, we investigate developers' efforts on repairing bugs caused by inappropriate TLUs. We notice that 32.53% of these bugs can be fixed by only modifying the build scripts (which we call TLU-build bugs), 20.48% of them can be fixed by merely modifying the source code (which is called TLU-code bugs), and 16.87% of them require modifications in multiple sources. TLU-build bugs and TLU-code bugs as the two most popular types, are explored more by us.

For TLU-code bugs, we summarize the common ways used to fix them. Furthermore, we study whether current repair techniques can fix TLU-code bugs efficiently. For the 14 TLU-code bugs that cause test failures and runtime failures, the study shows that existing automated program repair tools can only work on 6 of the 14 investigated bugs. Each of them can only fix a limited amount of the 6 bugs, but the union of them can finally fix 5 out of 6 bugs.

For TLU-build bugs, by leveraging the knowledge from our study, we summarize common patterns to fix build scripts, and propose a technique to automatically fix them. Our evaluation shows the proposed technique can successfully fix 9 out of 14 TLU-build bugs.

Acknowledgments

I first want to thank my supervisor, Prof. Lin Tan, for her patience and suggestions at every stage of my research journey. She acts as a role model for her students from many aspects, such as her research achievements, presentation skills, and work efficiency. Lin is also willing to share her experiences of her different life stages with us so that we can make wiser choices. I have already and will continue benefiting from what I learned from her.

I would also like to thank Prof. Meiyappan Nagappan and Prof. Patrick Lam, for spending time reviewing my work and giving their invaluable comments.

Many thanks to my research group members, who always help me during my graduate study. I still remember Jinqiu helped me settle down and drove me to buy daily necessities when I first arrived in Waterloo. She is also a mature researcher who assists me a lot. Taiyue as my first partner, was tolerant of my innocence and gave me so many helpful feedback. Alex came to our group at the same time with me. We shared our experience with each other as we always stepped into a new stage at the same time. Song is more like a mentor. He discussed different ideas with me to inspire my research. When I finished my thesis draft, he thoroughly read and helped polish my thesis. I also enjoy the time with other group members including Thibaud, Edmund, Xinye, and Yuan. We took courses, discussed research questions, and hung out together. All of these are precious memories for me.

Finally, I would like to express special thanks to my family and friends. Thanks to my parents, who always love me and support me; thanks to my friends, who share joys and sorrows with me. Without their support, I cannot finish my study successfully.

Dedication

This thesis is dedicated to the ones I love and the ones who love me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In modern software development, the appropriate utilization of third-party libraries undoubtedly alleviates repetitive work of project developers, so that they can focus on core function development. Thanks to the blossoming of open-source software development, there exist a large number of third-party libraries which cover various kinds of functionalities and are suitable for different development environments.

In this thesis, the developer who imports and uses the third-party libraries is called a *client*, and the developer who is responsible for developing and improving the third-party libraries is called a *supplier*. Clients upgrade a third-party dependency by changing the library version from an older version to a newer version, note that the terms—"older version"/"newer version"—used in this thesis mean that a version used by clients in an older date/a newer date.

When a library is used by clients, the suppliers are also working on improving updating their product out of these reasons: (1) the suppliers fix some vulnerabilities or bugs existing in current library version; (2) some modules of the library are rewritten or refactored, some old APIs are deleted or integrated, and new functionalities are introduced; (3) other external dependencies imported by the library have changed so that the library has to change itself. Therefore, newer versions will be released continuously, which can lead to multiple issues, like backward compatibility [52], API deprecation [53], or license change [14, 58].

Previous work has shown that 81.5% of the studied projects still keep their outdated dependencies [28]. However, sometimes clients have no choice but to do the version upgrade because of some reasons, for example: (1) the latest version fixes a substantial bug or vulnerability which could affect their projects; (2) they need to use a new functionality

that only exists in the newer version; In Junit version 4.11[1], clients updated the library Hamcrest[2] to version 1.3 in order to use a new feature of Hamcrest 1.3. (3) they want to update a library in their projects to a more stable version; (4) they just want to keep their dependencies up-to-date.

Sometimes suppliers go to great lengths to avoid compatibility bugs at the client end when they release a newer version of a library. In this thesis, we only care about library changes that break clients' projects, since other changes will transparently support upgrades.

1.1 A Study of Third-party Library Upgrade Bugs

An inappropriate Third-party Library Upgrade (TLU) can bring in diverse problems, such as buggy source code, configuration errors, or obsolete tests, which may affect different aspects of a program. We use *TLU bugs* to present bugs caused by TLUs. This thesis presents an empirical study on TLU bugs.

Our empirical study finds that TLU bugs are fixed in various sources of a program, such as source code, test code, and build scripts. In terms of different modified sources in which TLU bugs can be fixed, we use *TLU-code bugs* to denote bugs that can be fixed by merely modifying the source code, and *TLU-build bugs* denote bugs that can be fixed by changes only in build scripts. Also, there are some TLU bugs that need fixes in other sources, even across multiple sources (16.87%). Our study starts with all types of TLU bugs, and we find TLU-code bugs (20.48%) and TLU-build bugs (32.53%) are the two most popular types. So we then zoom in on TLU-code bugs (RQ4, RQ5) and TLU-build bugs (RQ6a, RQ6b) separately.

This study is conducted on a collection of 8,952 open-source Java projects from GitHub and 304 Java projects on ASF JIRA systems. We find 83 TLU bugs in total.

Our study addresses the following research questions:

RQ1: Why do clients upgrade a third-party library?

The motivation of studying RQ1 is as follows. Prior studies are proposed on clients' responses to library changes. Robbes et al. [53] only focus on API deprecation caused by library changes, and study how clients react to the API deprecation. The work of Kula

[1] `https://github.com/junit-team/junit4/blob/master/doc/ReleaseNotes4.11.md`
[2] `http://hamcrest.org/JavaHamcrest/`

et al. [28] also studies the similar topic. Their study is more related to this thesis. They find that developers are less likely to migrate their library dependencies, and analyze the reasons why developers do NOT update a library. These studies mainly show clients' slow response to library changes, and analyze reasons behind the reluctance.

However, RQ1 investigate the opposite direction. Although clients are less likely to upgrade libraries in most cases, sometimes they have to conduct the upgrades. RQ1 wants to answer under which situation clients need to update a library.

To answer RQ1, we choose 14 popular third-party libraries and collect version change histories about the 14 libraries from 8,952 GitHub Java projects. Then we analyze these dependency upgrade histories, and summarize reasons for these upgrade actions. We find that clients' reasons for upgrade third-party libraries fall into seven categories: 1) clients are preparing for release (62.3%) and want to make the program's dependencies up-to-date; 2) clients update other dependencies and the older version is not compatible with them (15.3%); 3) the newer version can help fix bugs/vulnerabilities in clients' program (7.1%); 4) the older version is not compatible with development kits (e.g., JDK/Android SDK) (6.7%); 5) clients want to use a particular feature not existing in the older version (3.9%); 6) the newer version fixes its bugs (3.6%); 7) the older version is unavailable (1.1%).

RQ2: What are the symptoms of TLU bugs?

Clients choose to upgrade their dependencies for different reasons (answered by RQ1). Some upgrades are urgent, and push clients to conduct the upgrade in a hurry, which can lead to inappropriate TLUs. Therefore, we use RQ2 to study the effects of inappropriate TLUs.

To answer RQ2, we first need to find TLU bugs. We find 83 TLU bugs through two ways: 1) analyzing JIRA issues; 2) analyzing commit messages. Towards the two ways, we crawl issues of 304 Java projects found in ASF JIRA issue system, and commits from top 1,000 projects of the 8,952 GitHub projects. Compared with the TLU bugs found through JIRA issues, the bugs found through commit messages lack detailed bug information. Therefore, we choose 62 TLU bugs found through JIRA issues as the dataset of RQ2. We manually analyze text contents (issue summary, description, and discussion) of these TLU related issues, to study the symptoms of TLU bugs. We find that inappropriate TLUs can introduce failures that are detectable statically (compilation failures) and dynamically (test failures and runtime failures).

RQ3: What are clients' efforts on fixing TLU bugs?

When an inappropriate TLU triggers a TLU bug, clients are dedicated to fixing it. We propose RQ3 to investigate clients' efforts on fixing TLU bugs.

For the 83 TLU bugs collected in RQ2, we collect their bug fixes from patch attachments on ASF JIRA systems, and code changes in GitHub commits. Then we manually inspect these bug fixes. We find clients' bug fixes mainly focus on four sources, i.e., source code (36.14%), build scripts (42.17%), test cases (22.89%), and version switch (9.64%). Besides, other actions may be conducted (7.23%), such as waiting for suppliers of the third-party library to make changes.

RQ4: What are the common ways for clients to fix TLU-code bugs?

Inspecting TLU bug fixes also inspires RQ4, i.e., are TLU bugs fixed according to some common ways? Since bug fixes conducted in different sources always have diverse characteristics, it is hard to generalize common patterns for all of them. Besides, bug repair in source code is a most popular research topic at present [27, 30, 36, 49, 50, 54, 64]. Therefore, we zoom in on TLU-code bugs to study this question.

We examine 17 TLU-code bugs from the 83 TLU bugs, and find 9 of them are fixed by four ways: 1) adding NULL value check for variables; 2) deleting a statement; 3) Moving a statement to another location; 4) replacing a statement with another one.

RQ5: Are the state-of-the-art automated program repair tools effective for repairing TLU-code bugs?

The motivation of RQ5 is that automated program repair has been studied by many researchers. Various automated repair tools are proposed to ease developers' debugging work. However, the performance of these tools cannot be thoroughly evaluated due to the lack of dataset. As TLU-code bugs have not been studied before, letting these repair tools fix TLU-code bugs is a good way to evaluate their performance on different types of bugs. Out of this object, we propose RQ5.

We only focus on 14 TLU-code bugs that cause test failures and reproducible runtime failures because existing automated program repair tools focus on the two types of failures. We reproduce 6 of the 14 TLU-code bugs for the evaluation. As for the other 8 bugs, their bug fixes involve big code changes or code changes at multiple locations, they cannot be solved by existing repair tools. Then we select four representative repair tools: GenProg [31], PAR [27], SPR [35], and Genesis [34], to evaluate their performance on fixing TLU-code bugs practically or theoretically. Our evaluation indicates that existing tools can only work on a limited number of TLU-code bugs (6 out of 14). Moreover, for the 6 bugs they can work on, each tool can only fix a small subset of them, but the union of these subsets can finally fix 5 out of the 6 TLU-code bugs. The other one bug cannot be fixed by any of the tools.

1.2 LibugFix

In our comprehensive study, we also observe a significant percentage (32.53%) of TLU bugs are TLU-build bugs, but there are few studies on automatically fixing buggy build code. We find there are several common patterns to fix the build scripts. These patterns are simple and easy to be implemented automatically. Therefore, we propose a technique named Third-party Library Upgrade Build Bug Fix (LibugFix). LibugFix automates the patterns found in our observation, to fix TLU-build bugs that cause compilation failures. TLU-build bugs can cause failures that are detectable statically (compilation failures) and dynamically (test failures & runtime failures). In this thesis, we only focus on fixing TLU-build bugs that cause compilation failures, and leave fixing TLU-build bugs that cause test failures and runtime failures to our future work.

LibugFix addresses the research questions:

RQ6a: What are common patterns for fixing TLU-build bugs?

RQ6b: How effective is LibugFix in fixing build scripts?

To generalize common patterns of fixing TLU-build bugs, we consider building scripts in the forms of .xml files, .properties files, and .classpath files, as these files' features are relatively similar. We use 14 TLU-build bugs that cause compilation failures to evaluate LibugFix, these bugs are all collected in RQ2. LibugFix can successfully fix 9 out of 14 bugs.

1.3 Contributions

This thesis makes the following contributions:

- We collect a dataset of TLU bugs. The dataset includes 62 bugs extracted from ASF issue tracking system, and 21 bugs from GitHub projects. For each collected bug, its related dependency upgrade information, problems that caused by TLU, and bug fixes are listed. To the best of our knowledge, we are the first to present a bug dataset of this type.
- We conduct a comprehensive study on TLU bugs. We manually analyze the bug fix of each TLU bug. We find that TLU bugs are fixed in four sources: modifying the source

code, fixing build scripts, updating obsolete tests, and switching the current dependency version to a different one. Moreover, 16.87% of these bug fixes involve modifications across multiple sources.

- We study the performance of state-of-the-art automated program repair tools on fixing TLU-code bugs. This study evaluates GenProg, PAR, SPR, and Genesis, which are representatives of generate-and-validate patch generation tools. The evaluation shows that for TLU-code bugs whose bug fixes involve big code changes or code changes at multiple locations, none of these tools can handle them. For TLU-code bugs whose bug fixes only involve code changes at a single location, every evaluated repair tool can only cover a limited aspect of them, but the union of these tools has potential to fix 5 out of 6 bugs of this type.

- We propose a technique that can successfully fix TLU-build bugs that causes compilation errors. For the 14 TLU-build bugs in our dataset, 9 of them can be fixed successfully.

Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 discusses the related work. Chapter 3 describes the data source of our empirical study. Chapter 4 shows our empirical study results. Utilizing findings from Chapter 4, Chapter 5 presents LibugFix and its fix results. Chapter 6 describes the threats of our work. Chapter 7 presents our findings and gives future work.

Chapter 2

Related Work

2.1 Software Evolution

Previous studies give comprehensive investigations on software evolution from different aspects [2, 7, 14, 17, 19, 25, 26, 29, 32, 47, 58, 61], including software license evolution [14, 58], API migration [29], component refactoring detection [17, 61], and so on. Several aspects related to this thesis are described specifically.

One related perspective focuses on studying issues caused by evolution of project dependencies [6, 15, 52]. Bavota et al. [5] observe the evolution of the Java subset of the Apache ecosystem, and analyze how an upgraded dependency impacts other related projects. Kula et al. [28] perform an empirical study to investigate the developers' reactions to library upgrade. Our study focuses on effects of library upgrades on library users' projects. Dietrich et al. [15] inspect evolution problems in Java programs caused by library upgrades. They focus on system runtime failures due to API changes in libraries that evolve independently, while our work is not limit to API level.

Another related aspect is studying API stability in software evolution [39,53]. Robbes et al. [53] conduct an empirical study on ripple effects of API deprecation caused by framework or library changes in a Smalltalk ecosystem. Their study shows that many deprecated methods and classes cause ripple effects, and the clients have to adapt to the dependencies they use. Robbets et al. focus on API level, and limit their work on a closed software ecosystem. But we are interested in any effects resulting from library upgrades, and we collect data from both Apache software foundation and GitHub open-source projects.

In addition, Pinto et al. [47] propose an approach and implement it in a tool for helping study and understand how test cases evolve in reality. The study indicates that test repair

is just one possible way to solve problems in test-suite evolution, while most changes involve refactorings, deletions, and additions of test cases. Moreover, test repairs tend to be complex, which makes them hard to be achieved automatically. Our study also touches this topic. We find that inappropriate third-party library upgrades may introduce obsolete tests, which needs to be repaired properly. Obsolete test repair is an interesting topic and we leave it to future work.

Towards these revealed evolution problems, previous studies propose various techniques from different emphases, to support library upgrades.

2.1.1 Incompatibility Detection

Dig et al. [16] combine syntactic and semantic analyses together to detect refactoring in evolving components. Abate et al. [1] utilize dependency analysis to predict upgrade failures.

More studies are proposed on API level to solve incompatibilities during software evolution. Taneja et al. [55] present a technique and its supporting tool *RefacLib* to automatically detect API refactorings. *RefacLib* firstly uses syntactic analysis to quickly detect refactoring candidates, then uses heuristics to refine the results. Raemaekers et al. [52] propose a way to measure interface and implementation stability through analyzing historical values of metrics, weighted by the times methods, classes or packages that are being used. They use historical version information to evaluate backward compatibility.

Besides, API misuses detection techniques [33, 42, 45, 57, 59] can also be used to detect bugs introduced by API changes during library upgrade. Gabel et al. [18] firstly present a dynamic way to automatically learn and enforce properties. Pradel et al. [48] combine the dynamic analysis and a static checker of API usage constraints to detect illegal API usage without human-written specifications. Recently, there are many studies [3, 24, 68] that are proposed to detect API misuses in dynamic languages.

Although many previous studies propose techniques to detect problems introduced by evolving components, our study still finds bugs resulting from library upgrades.

2.1.2 Library Client Adaptation

- **Dependency Level**

  Chow et al. [8] firstly assume that standard library change specifications are provided by the suppliers. Then they propose an approach to semi-automatically help the

library clients update their code with the change information. However, Xing et al. [63] think the documents provided by the library developers are incomplete. So they use a differencing technology on the old and new library versions to catch up the change, then use the change to make recommendations about likely ways to update the library.

- **API Level**

  The third-party library clients often encounter deprecated or updated API usage after updating to a newer library version. Hence research about API usage adaptation is also related to our work. Henkel et al. [21] present a tool named *CatchUp!* to semi-automatically help clients update their software components by capturing and replay the API refactoring processing. SemDiff [9] provides API replacement recommendation for errors that happen in compilation time. SemDiff suggests method calls by analyzing the source code history of the evolved component to figure the removed and newly added methods. Nguyen et al. [43] propose a graph-based approach called *LIBSYNC* to help API usage adaptation by learning from successful adaptations of other projects which have already updated to a new library version. Hora et al. [23] propose a tool named *apiwave*, which keeps track of API popularity and migration to address API backward-incompatibility.

  In addition, general API usage recommendation techniques are also helpful for adapting evolved method calls. Zhong et al. [70] introduce a API usage mining framework tool, *MAPO*, to automatically mine API usage patterns, then *MAPO* uses these patterns to provide API usage recommendation. Zhang et al. [67] give an approach named *Precise* to automatically recommend API parameters. *Precise* mines the existing source code, and abstract usage pattern representation for each API. Then it gives recommendations by finding the abstract usage patterns in similar contexts.

These studies all focus on source code adaptation, while our proposed technique focuses on fixing build scripts of Java projects.

2.1.3 Obsolete Test

In our empirical studies, we find obsolete tests are a noticeable problem in TLUs. 22.89% of inappropriate TLUs introduce obsolete tests. Due to clients' inattention, or because suppliers do not even describe all the changes in their release notes, the clients cannot catch every change made in the newer version of the third-party library. Therefore, TLU will

cause the current test cases to expect an obsolete behaviour which has changed in a newer library version, resulting in a failure. Tansey et al. [56] focus on annotation refactoring. Their work can infer general composite refactorings to help upgrade unit testing code. Hao et al. [20] adopt the Best-first Decision Tree Learning algorithm to train a classifier, and use it to identify whether the software failure cause lies in the functional source code or the test cases. Herzig et al. [22] analyze tens of millions of individual test steps, using association rule to identify patterns which are unique to false test alarms.

2.2 Fault Repair

Research on fault repair is another topic related to this thesis. Fault repair work includes repairing bugs existing in product code and repairing test suite.

2.2.1 Source Code Repair

Automated program repair has been studied for a long time. Researchers mainly focus on two fields: generate-and-validate (G&V) techniques [13, 27, 30, 36, 46, 49, 50, 54, 64] and semantic-based repair techniques [40, 41, 44, 65]. G&V techniques often locate suspicious location based on test executions, generate a list of candidate patches, then evaluate these candidates with test cases. Debroy et al. [13] propose a mutation technique to generate patches. Perkins et al. [46] introduce a tool named ClearView. ClearView firstly learns invariants from normal and erroneous executions. Then it finds correlated invariants that characterize normal and erroneous executions, according to which it generates a list of candidate. It finally decides the most successful patch based on the continued execution. Semantic-based repair techniques use symbolic execution and constraint solvers (e.g. SMT solver) to conduct patch synthesis, e.g. SemFix [44], DirectFix [40], Angelix [41], and Nopol [65]. In this thesis, we focus on G&V techniques.

Recently, a lot of G&V approaches are proposed from different aspects. GenProg [31] uses genetic programming to generate patches. GenProg as a pioneer work, inspires many follow-up research work. PAR [27] also uses genetic programming, but it manually inspects human-written patches, and summarizes common patterns which can be applied to modify the buggy source code. AE [60] cares about the performance of genetic programming (e.g., GenProg), it formalizes a cost model in terms of test executions, which can efficiently reduce the search space. RSRepair [49] shares the same operators with GenProg, but it uses random search instead of genetic programming. SPR [35] proposes a novel staged

program repair approach. It firstly defines parameterized transformation schemas, then uses target value search to decide whether a transformation schema can produce a successful patch. Furthermore, it generates a successful repair based on a condition synthesis algorithm. Using SPR as the baseline, Fan et al. then propose Prophet [36]. Prophet uses machine learning techniques to learn from successful human-written patches. Based on the knowledge, it re-orders the patch candidates generated by SPR, so that correct patches are ranked highly.

These G&V repair techniques have some limitations. First, these techniques such as GenProg assume that large programs contain the seeds of their own repair, so they use ingredients from current fields (i.e., current program, current class, or a pre-defined context) to generate patch candidates. Therefore, their patch search spaces are limited, and a correct patch can be only generated if the ingredients can be found in existing source code. Martinez et al. [38] conduct an empirical study to evaluate this fundamental assumption, this study is conducted on line level and token level. Barr et al. [4] study whether a fix can be constructed from historical fixes. They look into 15,723 commits from 12 Java projects and find 30% of the commit elements can be found within the same file, which means the remaining 70% involve ingredients from external sources. Another limitation is that current code transforms are pre-defined, which can only cover a small subset of correct bug fixes. Genesis [34] is a novel system which can automatically infer code transforms from existing successful human patches. But it can only handle three types of bugs in Java programs: null pointer, out of bounds, and class cast defects. These limitations caused a limited patch search space. In our evaluation, we focus on bugs introduced by inappropriate TLUs. In term of repairing this type of bugs, we find some correct bug fixes do not exist in current search space, because introducing a newer version of the dependency will also bring in new components. These new problems brought by a newer version of third-party dependency cannot always be solved with the current code redundancies. Therefore, enlarging the search space is an interesting research problem.

2.2.2 Test repair

Different test repair techniques are proposed to fix broken test cases in the software evolution [62, 66, 69]. Daniel et al. [12] present a repair suggestion tool named ReAssert to help developers' fix failed unit tests. Their technique considers both the dynamic analysis and static analysis. Daniel et al. then utilize symbolic execution [11] to improve the ReAssert, and add new repair strategies [10] that enable it to handle assertThat assertions.

As the noticeable obsolete test problem brought by inappropriate TLUs. Test repair is also a promising research area for TLU.

Chapter 3

Data Sources

To answer the research questions, we collect data on a collection of 8,952 open-source Java projects from GitHub and 304 Java projects on ASF JIRA systems. Figure 3.1 shows the data sources for the research questions.

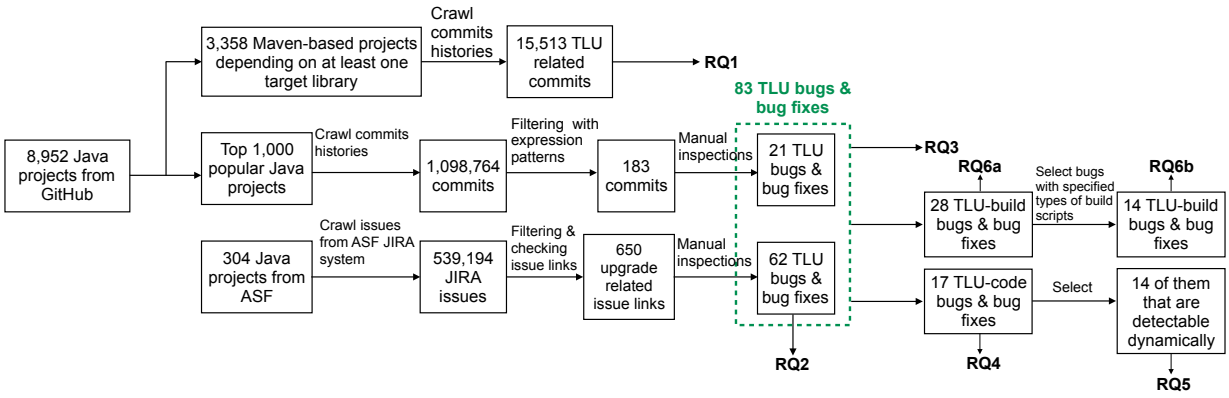

Figure 3.1: Data sources of research questions.

In summary, for **RQ1**, we select 14 target third-party libraries (see Table 3.2), and extract 3,358 Maven-based Java projects from the 8,952 GitHub projects as the subject projects of **RQ1**. The selected 3,358 projects depend on at least one of the 14 libraries. Then we examine change histories extracted from the 3,358 Maven-based Java projects about the 14 libraries (**RQ1**). Next, we study 83 TLU bugs, i.e., 21 found through GitHub

commits of top 1,000 popular Java projects from the 8,952 GitHub projects, and 62 found through JIRA issues of the 304 ASF java projects. For the 62 bugs with detailed JIRA issues, we investigate all their issue summaries, descriptions, and discussions to identify the symptoms of TLU bugs (**RQ2**). In addition, we examine patches on the 83 TLU bugs to see clients' efforts on fixing TLU bugs (**RQ3**). Furthermore, according to different sources in which TLU bugs are fixed, we zoom in on two types of TLU bugs, i.e., *TLU-code bugs* (whose fixes are *only* in source code), and *TLU-build* bugs (whose fixes are *only* in build scripts). We inspect 17 TLU-code bugs and their fixes for **RQ4**, and evaluate existing repair tools on 14 of them (**RQ5**). We then focus on building LibugFix for TLU-build bugs (**RQ6**).

3.1 Extracting Third-party Library Version Change History (RQ1)

To study the motivations of upgrading libraries, we examine the commit messages written by clients when they conduct TLUs, as clients may leave reasons in the commit messages. Therefore, we start with 14 popular libraries. Then, we find 3,358 Maven-based Java projects that depend on at least one of the 14 libraries from the 8,952 GitHub projects. Next, we extract all the TLU related commits based on the 14 selected target libraries. Finally, we check each commit message and extract the motivations behind the upgrade action.

Selecting Target Library: Considering the large amount of third-party libraries on which a project depends, we cannot study all the existing third-party libraries. Therefore, we first pick out some representative libraries as our research targets. Many previous studies have been conducted to find popular and important third-party libraries for Java projects, such as [51] and [29]. Qiu et al. [51] list the up-to-date top 20 popular Java libraries by analyzing data from GitHub between 2014/12/29 and 2014/12/31. We selected 14 libraries from the 20 libraries out of our interests in functionality-oriented libraries (See Table 3.2). As 8 of the 20 libraries all provide testing or logging functionalities used in testing, e.g., commons-logging and slf4j, we only take log4j and its successor logback into consideration.

Subject Project: Since a project with a longer development history always has more dependency upgrade actions, we select only projects that have at least one year of version history. Also, we only pick non-forked projects with licenses to avoid repetitive and informal

projects. We choose popular projects that earn at least five stars on GitHub. According to these rules, we collect 8,952 popular projects in total from GitHub.

Extracting TLU related commits: We currently only focus on Java Maven-based projects, which enable us to get the dependency relationship from the "pom.xml" file (Figure 3.2). We consider a commit TLU related if the commit participates in a version change. We present a version change as $V < v1, v2 >$, $v1$ means the older version, and $v2$ means the newer version, namely the upgraded version. Note that $v2$ may be a version that released before $v1$. For these 14 target libraries, we crawl and analyze pom.xml files of the 8,952 projects, and get 3,358 projects which depend on at least one of the 14 target libraries. Detailed information of these projects is listed in Table 3.3. The number of extracted TLU related commits for each target library is listed in Table 3.2.

Inspecting TLU related commit messages: After extracting the change history of all projects, we examine the whole change history files, and summarize the reasons behind dependency upgrades. Since it is prohibitively expensive to manually examine all TLU related commit messages to identify library upgrade reasons, we use keyword search to help filter out TLU related commits without specifying upgrade reasons, then manually inspect the remaining TLU related commit messages and classify upgrade reasons. Although this keyword-based approach may miss some commits that specify upgrade reasons in unusual ways, it is our best effort.

We first manually examine TLU related commit messages for the first five target libraries, and summarize keywords in commit messages that are usually relevant to specifying upgrade reasons (Table 3.1). Then we use these keywords to filter out those TLU related commits without reasons. Next, we manually inspect the remaining commit messages to summarize upgrade reasons. We only focus on commit messages written in English in this thesis.

Table 3.1: Keywords used to identify TLU related commits that specify upgrade reasons.

Keywords used	fix, repair, bug, error, vulnerab, exception, fail, brok, break, build, compil, chang, new, feature, function, method, unavailabl, prepar, releas, compatibl, deprecat, unavailabl

Table 3.2: Target third-party libraries. In a Maven-based project, the combination of groupId and artifactId can identify a library.

Rank	Library name (GroupId)	Library name (ArtifactId)	# TLU related commits
1	log4j	log4j	1,525
2	com.google.guava	guava	2,226
3	commons-io	commons-io	1,393
4	commons-lang	commons-lang	924
5	javax.servlet	servlet-api	702
6	org.springframework	spring-context	1,485
7	org.apache.httpcomponents	httpclient	1,265
8	ch.qos.logback	logback-classic	1,312
9	commons-codec	commons-codec	906
10	com.google.android	android	154
11	org.springframework	spring-core	1,267
12	joda-time	joda-time	837
13	org.springframework	spring-webmvc	853
14	org.codehaus.jackson	jackson-mapper-asl	664

Table 3.3: Subject projects for extracting TLU related commits.

Projects creation dates	before 2014-12-31
Projects last update	after 2016-01-01
Project type	Java Maven projects
Project popularity	marked at least 5 stars on GitHub
# Target libraries included	>= 1
# projects	3,358

Table 3.4: Study subjects. For projects from ASF JIRA system, the investigated unit is the issue, and for projects from GitHub, the investigated unit is the git commit.

From	# Projects	# Files	LOC	# Units
ASF JIRA	304	362,534	40,136,072	539,194
GitHub	1,000	694,789	68,929,396	1,098,764
Total	1,304	1,057,323	109,065,468	1,637,958

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>1.4</version>
        </dependency>
        ...
    </dependencies>
    ...
</project>
```

Figure 3.2: An XML snippet example.

3.2 TLU Bugs and Patches Collection (RQ2 - RQ6)

To study RQ2 - RQ4, we need to collect TLU bugs and their fixes. One challenge of collecting TLU bugs is that it is hard to identify whether a bug is caused by TLU. Mostly the developers will not describe the cause of a bug. Hence, although we can use heuristic techniques to help filter out irrelevant information, the manual investigation is still essential at the final step to decide whether or not a bug is a TLU bug. It is hard for researchers to manually check all the 8,952 GitHub projects, so we select the top 1,000 popular Java projects from them. The popularity of a project is evaluated with the number of stars that is marked by other GitHub users.

Also, we find JIRA issues can help indicate TLU bugs thanks to the issue links existing between issues. Therefore, we choose the ASF JIRA issue system as another source to find TLU bugs. We inspect 304 ASF projects that are listed in the ASF JIRA system.

Specifically, we utilize Scrapy crawling framework[1] to collect dataset from Apache JIRA systems[2] and GitHub open-source community[3]. Both of them provide APIs[4] [5] to facilitate data crawling. Table 3.4 shows the projects we use in this thesis. Our manual study focuses on two kinds of units: JIRA issues and GitHub commits.

[1] https://scrapy.org
[2] https://issues.apache.org/jira/
[3] https://github.com
[4] https://developer.atlassian.com/jiradev/jira-apis/jira-rest-apis
[5] https://developer.github.com/v3/

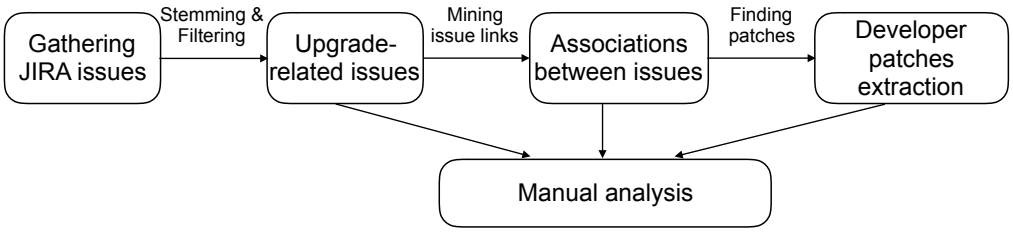

Figure 3.3: An overview of JIRA issues collection. We first crawl all the JIRA issues of Apache projects, then extract upgrade related issues from them. Next, we find all issues broken by these upgrade related issues, and collect fixes for these broken issues. We use both automatic techniques and manual inspection to conduct this data collection.

Collecting TLU bugs through JIRA issues: Apache JIRA, as an issue tracker, provides a lot of features which encode the relationships between issues. Figure 3.3 shows an overview of JIRA issues collection and analysis. First, we crawl all issues of Apache projects which release their source code in GitHub. A JIRA issue is composed of various fields[6], for our study we mainly focus on the fields of summary, key, description, affects version(s), fix version(s), Attachment(s), and issue link(s). Then, we utilize stemming techniques to stem the summary and description of the issues. After stemming, we filter out irrelevant issues and keep 47,222 issues which include key words "updat", "upgrad", "bump" in their summary or description.

To analyze the effects of upgrade issues, we extract the issue links[7] of each upgrade related issue. An issue link describes the association between two existing issues. In our study, we present an interested issue link as $L < I1, I2 >$, in which $I1$ means an issue that conducts a dependency upgrade action, and $I2$ means an issue that is broken by $I1$, namely is affected by the dependency upgrade action. We only pay attention to the issue link whose type is "breaks" or "is broken by". Figure 3.4 gives an example of issue links. This figure shows issue links related to a TLU related issue. The project HBASE upgrades its thrift dependency to version 0.9.2, which introduces a bug (HBASE-14162) in their project. In this way, we collect 650 upgrade related issues that breaks other issues.

Although we automatically filter out irrelevant issues, the remaining issues are not all about TLU related issues. For example, the issue ACCUMULO-4646 is flagged as a possibly TLU related issue, as its summary is: "Updates to INSTALL.md". However, this issue actually updated the project INSTALL.md file, which is not about upgrading

[6] `https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html`
[7] `https://confluence.atlassian.com/jiracoreserver073/linking-issues-861257339.html`

Figure 3.4: An example of issue links. The issue HBASE-14045 upgrades its dependency thrift to version 0.9.2, this TLU related issue breaks another issue HBASE-14162, and is related to issue HBASE-7972. In this work, we only care about issue links whose types are "breaks".

a third-party library. To mitigate the false positive, we further filter out these irrelevant issues with manual analysis. We finally collect 62 bugs through mining JIRA issues.

Collecting TLU bugs through GitHub commit messages: Unlike JIRA issue system, GitHub does not provide explicit features to describe associations between existing commits, which increases the difficulty of identifying the cause of a bug. We firstly crawl the whole commit histories of top 1,000 popular Java projects on GitHub. Similarly, we stem the commit messages with stemming techniques. After manually looking into around 1,000 commit messages, we generalize some common expression patterns to indicate whether a commit is to fix bugs introduced by TLU, and use these patterns to automatically extract 650 relevant commits. Table 3.5 lists the patterns that we use to extract TLU related commits. Then, we also conduct manual inspections to reduce the false positives. In total, we collect 21 bugs through analyzing commit messages. GitHub also provides an issue tracking system, which contains many useful information. We will explore this data source in the future.

Table 3.5: Commons Expression Patterns.

No.	Expression Pattern
1	fix/repair/handle ... after/because/as/since...upgrad/updat/bump
2	fail/error ... upgrad/updat/bump
3	cause/reason ... upgrad/updat/bump
4	upgrad/updat/bump ... introduc/caus/bring ... fail/error

Collecting developers' patch for TLU bugs: After obtaining TLU bugs through JIRA issues and GitHub commit messages, we can then collect patches for these bugs.

For bugs found through JIRA issues, we use two ways to find corresponding patches: for each issue link $L < I1, I2 >$, we first check whether patch attachments are available in $I2$, if not, we search the issue number of $I2$ in GitHub to find related commits.

For bugs found through GitHub commits, we will use the code change of the current commit as the patch.

3.3 TLU-code bugs (RQ4 and RQ5)

As TLU-code bug is the second popular type in TLU bugs, and existing automated program repair tools care more about TLU-code bugs. So we zoom in on TLU-code bugs in RQ4 and RQ5.

In total, there are 17 TLU-code bugs among the 83 TLU bugs. We examine the 17 bugs to find common fix patterns for RQ4.

For RQ5, most existing automated repair tools only consider bugs that are detectable dynamically, i.e., test failures and runtime failures, and they focus on fixing them by modifying the source code. We hence collected 14 TLU-code bugs of this type.

Also, most existing automated repair tools only conduct patch synthesis at a single location. Considering their limitation we hence select TLU-code bugs whose fixes satisfy constraints below:

1) the fix fixes a bug which can be reproduced by more than one failed test case;

2) the fix only modifies source code files (or modifies both source code and obsolete tests, as we can manually change the tests to be correct and keep the source code unchanged);

3) the fix only involves code changes at a single location;

4) the fix does not change code statements with special Java language features, such as Java annotation.

According to constraints above, we select 6 TLU-code bugs from the 14 bugs, and reproduce them to evaluate existing automated repair tools

3.4 TLU-build bugs (RQ6)

For the 83 TLU bugs that we collected, 18 bugs of them are TLU-build bugs that cause compilation errors, i.e., their fixes only involve modifications on build scripts. Table 3.6 lists the 18 TLU-build bugs. Build scripts are usually XML files (Maven-based or Ivy-based projects), .classpath files, .properties files, build.gradle files (Gradle-based projects), Bash scripts, or Python scripts. We manually inspect bug fixes of the 18 TLU-build bugs, and summarize common patterns of fixing TLU-build bugs (**RQ6a**). The results are shown in Section 5.1.

Table 3.6: TLU-build bugs that cause compilation errors. "GitHubProj" means projects from GitHub.

Project	# Bug	Suffix of fixed build scripts
ACCUMULO	1	.xml
BIGTOP	2	.py
HADOOP	5	.xml, .sh, .conf
HBASE	1	.xml
HIVE	4	.xml, .classpath, .thrift
ZOOKEEPER	1	.xml
SOLR	2	.xml, .properties
GitHubProj	2	.xml
Total	18	

Through studying RQ6a, we find these common patterns are mainly applied to fix build scripts whose types are .xml files, .properties files, and .classpath files. Therefore, we propose LibugFix to automatically repair build scripts that belong to the three types (**RQ6b**). There are 14 TLU-build bugs whose build scripts are .xml files, .properties files, or .classpath files. The 14 TLU-build bugs will be used as the dataset of LibugFix.

Chapter 4

Manual Study

4.1 RQ1: Why do clients upgrade a third-party library?

Commit messages contain useful information: they describe changes, and sometimes, reasons for these changes. We manually checked commit messages of the library-upgrade changes, and classified different types of reasons.

Table 4.1 shows the number of TLU commits with reasons why the clients upgrade the target libraries. Although we have a large number of projects with plenty of library-upgrade changes, the number of commits with upgrade reasons is not that big. We only have 9.5% of commits telling the reason. Fortunately, we still have 926 pieces of useful messages for our study.

Table 4.2 shows the distribution of the different reasons why clients choose to change the versions. We classify all the reasons into 7 different types: 1) new version with bugs fixed; 2) version with particular feature; 3) version unavailable; 4) prepare for releasing; 5) compatible with other dependencies/frameworks/platforms; 6) fix bugs/vulnerabilities of the project; 7) compatible with development kit (JDK/Android SDK).

"Prepare for release" is a more interesting phenomenon: before releasing a new version of the project, clients seem to always upgrade the versions of all dependencies to the newest ones. It seems a custom in software developing, as we found 62.3% TLUs specify this reason.

"Compatible with other dependencies/frameworks/platforms" means that the current version of the library will cause some problems when cooperating with other dependencies

Table 4.1: Upgrade commits with reasons. "# TLU related" means the number of total TLU related commits, "# TLU related with reasons" presents the number of TLU related commits that specify upgrade reasons.

GroupId	ArtifactId	# TLU related	# TLU related with reasons
log4j	log4j	597	73 (12.2%)
com.google.guava	guava	1,409	219 (15.5%)
commons-io	commons-io	664	82 (12.4%)
commons-lang	commons-lang	290	10 (3.5%)
javax.servlet	servlet-api	158	27 (17.1%)
org.springframework	spring-context	1,432	90 (6.3%)
org.apache.httpcomponents	httpclient	986	40 (4.1%)
ch.qos.logback	logback-classic	915	215 (23.5%)
commons-codec	commons-codec	521	15 (2.9%)
com.google.android	android	52	14 (26.9%)
org.springframework	spring-core	1,133	52 (4.6%)
joda-time	joda-time	508	13 (2.6%)
org.springframework	spring-webmvc	738	57 (7.7%)
org.codehaus.jackson	jackson-mapper-asl	300	19 (6.3%)
Total		9,703	926 (9.5%)

Table 4.2: Classification of upgrade reasons.

Type of reasons	Number	%
Prepare for release	577	62.3
Compatible with other dependencies/frameworks/platforms	142	15.3
Fix bugs/vulnerabilities of the project	66	7.1
Compatible with development kit (JDK/Android SDK)	62	6.7
Version with particular features	36	3.9
New version with bugs fixed	33	3.6
Version unavailable	10	1.1
Total	926	100

or running on some frameworks or platforms. So the clients decide to replace it with another version.

Table 4.3: Upgrades and downgrades. The combination of groupId and artifactId can identify a library. # Changes column shows the number of version changes we collect. These version changes are composed of upgrade version changes (4th column) and downgrade version changes (5th column).

GroupId	ArtifactId	# Changes	Upgrade (%)	Downgrade (%)
log4j	log4j	597	87.1	14.6
com.google.guava	guava	1,409	88.2	11.9
commons-io	commons-io	664	89.0	11.0
commons-lang	commons-lang	290	94.1	5.9
javax.servlet	servlet-api	158	87.3	12.7
org.springframework	spring-context	1,432	91.6	8.5
org.apache.httpcomponents	httpclient	986	91.6	8.4
ch.qos.logback	logback-classic	915	83.4	16.6
commons-codec	commons-codec	521	85.8	14.2
com.google.android	android	52	86.5	13.5
org.springframework	spring-core	1,133	93.1	6.9
joda-time	joda-time	508	89.8	10.2
org.springframework	spring-webmvc	738	94.9	5.2
org.codehaus.jackson	jackson-mapper-asl	300	91.0	9.0
Total		9,703	89.8	10.2

"Fix bugs/vulnerabilities of the project" means that clients change the version to fix bugs in their own projects.

"Compatible with development kit (JDK/Android SDK)" means the clients change the version of the library because the current version is not compatible with the JDK or Android SDK they use.

"Version with particular features" means that the clients choose to change (either "upgrade" or "downgrade") the version because they want to use some particular features only in the target version.

"New version with bugs fixed" is different from "Fix bugs/vulnerabilities of the project". The goals of the two reasons are different. For "New version with bugs fixed", suppliers release a newer library version, and this newer version fixes bugs existing in the older version. These fixed bugs may or may not cause problems for the clients' project at this time. Sometimes clients upgrade the library in advance to avoid potential troubles.

For "Fix bugs/vulnerabilities of the project", clients already encounter problems in their project due to the current library version, e.g., the behaviours of the older version are compatible with other dependencies. Clients want to fix the problems by upgrading the library.

"Version unavailable" is, just as its literal meaning, that the clients cannot use the current version of the library at the time of the TLU commit. In our observations, it may be because the current version has been removed from Maven Central Repository, or because there are no public download sources of this version now.

By comparison, we find that besides preparing for a version release (which is more like a custom in software development), the library version with fewer bugs is not that attractive for clients when deciding if it is needed to change the version. However, if the new version can fix problems in their own projects, like fixing bugs or ensuring the compatibility, a change will be more likely to be made.

Another interesting discovery is that the version changes are not always "upgrades", and sometimes clients may choose to "downgrade" the third-party libraries they use. Table 4.3 shows the number of upgrades and downgrades in our dataset. Although upgrades are the majority, "downgrades" still accounts for more than 10% of TLUs. This phenomenon relates to the frequent "revert" actions on GitHub. When faced with a problem which may be caused by TLUs, a revert is always conducted immediately to avoid troubles.

4.2 RQ2: What are the symptoms of TLU bugs?

We use this question to investigate how TLU bugs affect clients' projects. To answer this question, we investigate 62 TLU bugs collected through ASF JIRA issue systems, because they always have detailed issue reports and discussions. We do not consider TLU bugs collected through GitHub commit messages, because for the 21 TLU bugs from GitHub, the client often leaves a simple message, e.g., "fix problems caused by library upgrade". So it is hard to get clues from these commit messages.

For the 62 TLU bugs, we look into any available sources from issue description, discussion contents, and commits messages. We find that TLUs can introduce failures that can be detected statically or dynamically. "Detectable statically" column lists the number of failures detected statically. Failures detected statically means bugs that can be caught during compilation time, i.e., compilation failures. "Detectable dynamically" column lists the number of failures detected dynamically. A failure that is detectable dynamically can be a test failure, or a crash during runtime.

Table 4.4: The types of TLU bugs. "Detectable statically" column lists the number of failures detected statically, and "Detectable dynamically" column lists the nubmer of failures detected dynamically.

Project	# broken issues	Detectable statically (Compilation failure)	Detectable dynamically	
			Test failure	Runtime failure
Accumulo	10	3	7	0
Bigtop	4	4	0	0
Camel	1	1	0	0
Hadoop	20	9	6	5
Hbase	4	1	1	2
Hdfs	2	0	2	0
Hive	7	5	1	1
Jclouds	1	0	0	1
Kafka	1	0	1	0
Karaf	1	0	0	1
Oak	1	0	1	0
Ofbiz	2	0	0	2
Oozie	1	0	1	0
Pig	1	0	1	0
Rave	1	0	0	1
Solr	4	2	2	0
Zookeeper	1	1	0	0
Total	62	26 (41.94%)	23 (37.10%)	13 (20.97%)

Table 4.4 shows the composition of bug types introduced by TLUs. For the problems caused by inappropriate TLUs, 41.94% of them are detectable statically, and the failures found dynamically are composed of test failures (37.10%) and runtime failures (20.97%). Note that the number does not mean that inappropriate TLUs will lead to more failures that are detectable dynamically. In reality, it is easier for clients to detect compilation failures when they conduct a build action. Developers often fix them at once, but they do not leave any written records about this issue.

Indeed, one wonders why compilation failures exist at all. There are two possible reasons: (1) developers use continuous integration testing to check the program that passes local unit tests; (2) some compilation errors are not detected when developing, but happen on other platforms.

4.3 RQ3: What are clients' efforts on fixing TLU bugs?

In our study, we find clients often fix TLU bugs from four sources: modifying the source code, fixing build code, fixing obsolete tests, and avoid current dependency version. Table 4.5 describes the fix effort of clients.

Table 4.5: Fix efforts of clients. In column 4-8, the number outside the () means the number of bugs whose fixes involve modifying this source; the number inside the () means the number of bugs that can be fixed by modifying the single source.

Project	# broken issues	Multiple sources	Fix effort				
			Source code	Build scripts	Test cases	Version switch	Others
Accumulo	10	3	6 (3)	4 (1)	1 (1)	1 (1)	1 (1)
Bigtop	4	0	1 (1)	2 (2)	0 (0)	0 (0)	1 (1)
Camel	1	1	1 (0)	1 (0)	1 (0)	0 (0)	0 (0)
Hadoop	20	3	5 (3)	10 (8)	4 (2)	1 (1)	3 (3)
Hbase	4	1	2 (1)	2 (2)	1 (0)	0 (0)	0 (0)
Hdfs	2	0	1 (1)	0 (0)	0 (0)	1 (1)	0 (0)
Hive	7	0	0 (0)	5 (5)	1 (1)	1 (1)	0 (0)
Jclouds	1	0	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
Kafka	1	0	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
Karaf	1	0	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
Oak	1	0	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)
Ofbiz	2	0	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)
Oozie	1	0	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
Pig	1	0	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)
Rave	1	0	0 (0)	1 (1)	0 (0)	0 (0)	0 (0)
Solr	4	0	0 (0)	2 (2)	2 (2)	0 (0)	0 (0)
Zookeeper	1	0	0 (0)	1 (1)	0 (0)	0 (0)	0 (0)
GitHubProj	21	6	13 (7)	6 (4)	8 (4)	0 (0)	0 (0)
Total	83	14	30 (17)	35 (27)	19 (11)	8 (8)	6 (6)
Percentage	- 100%	16.87% -	36.14% (20.48%)	42.17% (32.53%)	22.89% (13.25%)	9.64% (9.64%)	7.23% (7.23%)

In the "Project" column of Table 4.5, we list all projects found from the ASF JIRA system and GitHub. As projects collected from GitHub are discrete across many different projects, it is hard to list all names of these projects. So we use "GitHubProj" to present projects from GitHub. "Source Code", "Build Scripts", "Test Cases", and "Version switch" present the modified sources for fixing TLU bugs. "Version switch" means clients choose

Table 4.6: Mean and five-number summary of fixing time for TLU bugs.

Fix time (day)	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
Multi-sources	34.6	0	1	3.5	12.3	227
Source code	53.8	0	0	3	11.8	277
Build scripts	57.3	0	1	6	64.5	386
Test cases	182.7	0	1	1	174	928
Version switch	24.3	0	1	1	3.5	182

to avoid current dependency version to fix the problem. A version switch may be: 1) reverting to its previous version; 2) changing to another different version. The number in the bracket means the number of bugs that can be fixed by modifying one single filed. "Multiple sources" (column 3) lists the number of bugs whose fixes need to modify more than one source. "Others" in this table means all other situations that are not covered by the first four sources. For example, this bug is not urgent, and clients find it is caused by a bug existing in the third-party library. So they just wait for the bug of the third-party library is fixed. Or they are still working on the bug until the day when we collected data.

The correct fixes for 16.87% of these bugs involve modifying multiple sources, for them, clients should have an eye on the global sources of the program, instead of focusing on the single source. Also, merely 20.48% of TLU bugs can be fixed by modifying the source code. In addition, 33.73% of TLU bugs involves changing build scripts, which is a significant part.

Table 4.6 shows the time spent on fixing bugs that involve modifications in different sources. The mean and five-number summary is given to evaluate the fixing time. Note that the fixing time cannot directly reflect the difficulty of bug fixing, because it is also related to the significance of a bug. For bugs fixed in multiple sources, they are always repaired very soon, which means these bugs have a significant influence on the program. For bugs that are repaired by merely modifying test cases, they always cannot draw clients' attention immediately. So this bug type always takes a longer time to fix.

4.4 RQ4: What are the common ways for clients to fix TLU-code bugs?

Focusing on the source code source, we examine 17 TLU-code bugs and summarize some common fixing ways for 9 of them. We found they are fixed according to four common ways: 1) add null check (NC); 2) deleting the buggy statement (DEL); 3) changing the

position of the buggy statement (MOV); 4) replacing the buggy statement with another statement, or replacing the value of variables/constant in the buggy statement with another one (REP). Table 4.7 presents in which way the 9 bugs are fixed. We use the case study to help explain these common ways.

Table 4.7: Common Ways to Fix TLU-code Bugs. ✓ means the bug is fixed in this way.

Bug	NC	DEL	MOV	REP
ACCUMULO-4077-4117	✓			
ACCUMULO-898-3342		✓		
HADOOP-12767-11859	✓			
HADOOP-9623-10589	✓			
HBASE-13339-13574				✓
HDFS-9080-9187				✓
OAK-3996-4018			✓	
Dropwizard-821174		✓		
tomahawk-android-8a40d9b		✓		
Total (9)	3	3	1	2

4.4.1 Case Study

- **NC**

```
1   private String getUserName(HttpServletRequest request) {
2  --  List<NameValuePair> list = URLEncodedUtils.parse(request.getQueryString(), UTF8_CHARSET);
3  ++  String queryString = request.getQueryString();
4  ++  if(queryString == null || queryString.length() == 0) {
5  ++    return null;
6  ++  }
7  ++  List<NameValuePair> list = URLEncodedUtils.parse(queryString, UTF8_CHARSET);
8      ...
9   }
```

Figure 4.1: HADOOP-12767-11859.

Figure 4.1 presents an example of NC.

HADOOP-12767[1] upgrades its httpclient dependency from 4.2.5 to 4.5.2, and httpcore dependency from 4.2.5 to 4.4.4. This upgrade causes test failure that method PseudoAuthenticationHandler throws a Null Pointer Exception (NPE). The reason is that

[1] https://issues.apache.org/jira/browse/HADOOP-12767

PseudoAuthenticationHandler method calls an API, URLEncodedUtils.parse (String, Charset), which is from the httpclient dependency. In httpclient 4.2.5, the parse() method can gracefully handle the first argument being NULL. However, after httpclient 4.4.4, it will throw an NPE if the first argument is NULL.

- **DEL**

```
1   public boolean isMaxFileSizeSettingSpecified() {
2     ...
3     getFilterFactories().stream().forEach(f -> appender.addFilter(f.build()));
4 --  appender.stop();
5     appender.start();
6     ...
7   }
```

Figure 4.2: Dropwizard-821174.

Figure 4.2 shows an example of DEL.

In this example, the appender of the program cannot be started after upgrading logback version from 1.1.3 to 1.1.6. We checked the source code of logback, and found that the start() method FileAppender.java changed in logback. In the version beyond 1.1.5 of logback, before starting an appender, it will conduct a collision check, which means that if an appender has already been used, it cannot be restarted. In the original source code the program, an appender always conducts a stop() action before a start() to make sure a starting appender does not be started again, even if the appender is initiated just before the start(). Before the upgrading, the "always stop()" action before start() does not cause problems. But after upgrading, as the collision check of logback beyond version 1.1.5, the appender.start() in line 5 cannot be correctly executed.

- **REP**

```
1   public Globber(FileSystem fs, Path pathPattern, PathFilterfilter) {
2     ...
3     this.filter = filter;
4 --  this.tracer = fs.getTracer();
5 ++  this.tracer = FsTracer.get(fs.getConf());
6   }
```

Figure 4.3: HDFS-9080-9187.

Figure 4.3 shows an example of REP.

HDFS-9080[2] updates htrace version to 4.0.1, which causes an NPE in HDFS-9187[3]. The motivation of this upgrade is that clients want to use new features from htrace 4.0.1,

[2]https://issues.apache.org/jira/browse/HDFS-9080
[3]https://issues.apache.org/jira/browse/HDFS-9187

in which package *org.apache.htrace.core* is introduced. Along this upgrade, clients also make a big code change in their program to coordinate with this upgrade. After the upgrade, an NPE will happen, because the tracer will be null if the filesystem is not constructed via FileSystem#createFileSystem. The best way to solve this problem is adding a new object "NullTracer" in htrace library to present the null tracer, but this depends on contribution of clients of htrace. Waiting for suppliers of htrace to solve this problem will waste lots of time, and leave uncertainty. Hence, clients change the way to get tracer, instead of get tracer from fs (which may be a null value), they get tracer from a pre-defined configuration file.

- **MOV**

```
1   private void setChildOrder() {
2     ...
3 ++  for (PropertyEntry property : bundle.getPropertyEntries()) {
4 ++    String name = createName(property.getName());
5 ++    try {
6 ++      ...
7 ++    } catch (Exception e) {
8 ++    }
9 ++  }
10
11    //code to prepare mixinTypes
12    Set<String> mixins = newLinkedHashSet();
13    ...
14
15 --  for (PropertyEntry property : bundle.getPropertyEntries()) {
16 --    String name = createName(property.getName());
17 --    try {
18 --      ...
19 --    } catch (Exception e) {
20 --    }
21 --  }
22    return properties;
23  }
```

Figure 4.4: OAK-3996-4018.

Figure 4.4 shows an example of MOV.

In the OAK project, OAK-3996[4] upgrades its jackrabbit version from 2.11.3 to 2.12.0, which causes test failures (OAK-4018[5]). The newer version jackrabbit 2.12.0 modifies its jcr:mixinTypes property in *JackrabbitNodeState#createProperties()* method. In the Oak project, the jcr:mixinTypes is prepared before line 15. Then the mixinTypes property is exposed in *NodePropBundle#getPropertyEntries()* collection (line 15 - 21). This upgrade causes that the original jcr:mixinTypes property (defined by jackrabbit) **overrides** the

[4]https://issues.apache.org/jira/browse/OAK-3996
[5]https://issues.apache.org/jira/browse/OAK-4018

one prepared by the Oak project. The fix for this bug is simple, that is, moving the for statement (line 15 - 21) before the code that prepares the jcr:mixinTypes (line 11), in this way the wanted jcr:mixinTypes will not be overridden.

Table 4.8: Available transformations that different automated program repair tools can implement semantically. ✓means the tool is able to implement this transformation , △ means the tool implements a subset of this transformation.

Transformation	**GenProg**	**PAR**	**SPR**
Deleting the buggy statement	✓		✓[1]
Swapping the buggy statement with another statement existing in the program	✓	△[2]	
Inserting a statement before the buggy statement	✓		✓
Inserting an initialization for variables in the buggy statement		△[3]	✓
Inserting an if-guard statement whose condition expression is a constant related condition[4]			✓
Inserting a null checker for object references existing at the buggy location		✓	
Inserting a range checker to check whether an index exceeds upper/lower bounds		✓	
Inserting a class cast checker to check the castee is an object of the casting type if the buggy statement is a class-casting statement		✓	
Introducing a new control flow (return, break, goto) whose condition expression is generated from an abstract condition[4]			✓
Changing a branch condition by replacing condition expression with another existing condition expression		✓	
Changing a branch condition by concatenating current condition expression and a constant related condition[4] using && or ||			✓
Adding/Removing parameters of a method at the buggy location if this method has overloaded methods		✓	
Replacing a value (variable/constant/method) with another one in the suspicious statement		△[5]	✓

[1] SPR can delete a statement by inserting an if-guard, such as if (!(1)) {...}.

[2] PAR only swaps statements both existing at the buggy location.

[3] PAR only initializes parameters of a method call if there exists a method call at the buggy location.

[4] A constant related condition is in forms of (1), !(1), (variable == const), or !(variable == const), the pair of varialbe and const can be found during the program execution.

[5] For variable/constant replacement, PAR only replaces them when they are used as parameters of a method call if there exists a method call at the buggy location. For method replacement, PAR only replaces method with another existing one which calls the same parameters. PAR and SPR both replace variable with another one existing in the program.

4.5 RQ5: Are the state-of-the-art automated program repair tools effective for repairing TLU-code bugs?

We use 14 TLU-code bugs that cause test failures or runtime failures to evaluate this research question. For 8 bugs of them, their bug fixes involve big changes or code changes in multiple locations. Existing program repair tools are not able to generate this kind of patches for them. Therefore, we use the remaining 6 bugs listed in Table 4.9 to evaluate the performance of automated program repair tools.

Since GenProg is initially designed for bugs in C, in this thesis, we use jGenProg, which is a Java version of GenProg implemented by a program repair framework named Astor [37]. We run jGenProg to repair the 6 bugs, with the default setting same as Astor. For PAR, SPR, and Genesis, we conduct theoretical analysis. SPR is a repair tool for C programs. We want to evaluate it because it defines various code transforms. But we cannot find a Java implementation of SPR. PAR is a Java program repair tool, but the authors does not release their tool to the public. Genesis is an interesting repair approach for Java program. It can automatically infer code transforms from successful human-written patches, so we also want to discuss it in this research question. However, this tool is newly released, and it needs to be run under a virtual machine with a fixed configuration. It is hard to successfully deploy these bugs to the virtual machine. Due to time constraints, we do not run Genesis and leave it into future work.

We compare the code transforms that GenProg, PAR, and SPR can conduct during the patch generation processing. As Genesis generates patches by automatically inferring code transforms from existing code, we cannot list the fixed transforms that it can achieve. The comparison is given in Table 4.8.

Table 4.9 shows our evaluation results. We find that HDFS-9080-9187 cannot be fixed by any inspected tools. The correct fix for HDFS-9080-9187 involves replacing the buggy statement with another statement that does not exist in the program. This means the ingredients for this correct patch are not in the program. This problem cannot be handled by any of the evaluated tools.

For jGenProg, it generates two correct patches for Dropwizard-821174 and ACCUMULO-898-3342. The two bugs are both fixed by DEL. jGenProg generates a plausible patch for OAK-3996-4018, which also conducts a DEL. A plausible patch means a patch that is actually incorrect, but passes all the test cases because of test incompleteness. According to Table 4.8, we can see that GenProg cannot insert a null check, so it cannot fix HADOOP-12767-11859 and HADOOP-9623-10589 because their correct fixes are not in the search space.

Table 4.9: Current Repair Tools on TLU Bugs. "✓✓" means the tool generated a correct patch for the TLU bug, "△" means the tool generates a patch, but the patch is plausible. A plausible patch means a patch passes all the test cases, but is incorrect. "✓" means the tool could, in principle, generate a patch that fix the TLU bug. "-" means the tool cannot generate the correct fix theoretically.

Defect	**GenProg**	**PAR**	**SPR**	**Genesis**
HDFS-9080-9187 (REP)	-	-	-	-
Dropwizard-821174 (DEL)	✓✓	-	✓	-
ACCUMULO-898-3342 (DEL)	✓✓	-	✓	-
HADOOP-12767-11859 (NC)	-	-	-	✓
HADOOP-9623-10589 (NC)	-	-	-	✓
OAK-3996-4018 (MOV)	△	-	-	-
Total	2	0	2	2

For PAR, although it does not release their tool, it makes their fix templates and generated patches publicly available[6]. We inspect the fix templates and generated patches, and find that PAR cannot fix the 6 inspected bugs. It seems that PAR may fix HADOOP-12767-11859, and HADOOP-9623-10589. However, PAR only checks object references existing at the buggy location. That is, if a parameter is not referenced, it does not check whether it is null. The objects that need NC in HADOOP-12767-11859 and HADOOP-9623-10589 are both not referenced, so PAR cannot fix them.

For SPR, it cannot fix HADOOP-12767-11859, HADOOP-9623-10589, and OAK-3996-4018 as it cannot conduct corresponding code transforms to generate correct fixes.

Genesis takes a different approach from the first three tools. It can only handle three types of bugs: null pointer, out of bounds, and class cast defects. Also, Genesis automatically infers code transforms for patch generation. Considering the correct fixes for HADOOP-12767-11859 and HADOOP-9623-10589 are both adding a null check for an object in the buggy statement. We inspect the patches generated by Genesis for null pointer bugs, and find they are similar with patches for HADOOP-12767-11859 and HADOOP-9623-10589. Therefore, Genesis may be able to generate correct fixes for these two bugs, at least they exist in Genesis's search space.

[6] `https://sites.google.com/site/autofixhkust/home/fix-templates`

Chapter 5

LibugFix

5.1 RQ6a: What are common patterns for fixing TLU-build bugs?

A large program often includes multiple sub-modules, these sub-modules share a set of dependencies, but also manage their own dependencies. To handle this, every sub-module has its own build scripts, at the same time inheriting its parent build scripts. When updating a third-party library, clients often do not handle the complex dependency relationships very well. Therefore, in Table 5.1, we summarize three common fix patterns used to repair build scripts.

Pattern 1 focuses on the problems of inconsistent upgrades. When updating a third-party library, all corresponding should also be updated. However, clients often forget to update all features related to the third-party library, which will cause conflicts. For example. in the ZOOKEEPER-2378[1] issue, clients upgrade their ivy dependency version from 2.2.0 to 2.4.0. However, they forgot to update the corresponding ivy version in its contrib sub-module. The issued ivy version in the build script of contrib is still 2.2.0, which causes a build failure.

The fix for Pattern 1 is to align the versions of dependencies related to L to the newer version.

Pattern 2 solves the problems resulting from the incomplete upgrades. This problem often appears when upgrading a version from null to a version number (introducing a new

[1] `https://issues.apache.org/jira/browse/ZOOKEEPER-2378`

Table 5.1: Common patterns for fixing build scripts.

No.	Pattern	Description	Example
1	Aligning attributions related to the upgraded dependency with the latest version	This pattern can fix a bug which is caused by incomplete or inconsistent upgrade	<dependency> <groupId>group_id</groupId> <artifactId>artifact_id</artifactId> - <version>order_version</version> + <version>latest_version</version> </dependency>
2	Removing/Adding extra/duplicated dependencies	This pattern can fix a bug which is caused by duplicated/missing dependencies introduced by library upgrade	- <dependency> - <groupId>group_id</groupId> - <artifactId>artifact_id</artifactId> - <version>1</version> - </dependency>
3	Reverting attributions of the dependencies which belong to the same group ID with the upgraded dependency	This pattern can fix a bug that the dependencies originally sharing the same attributions with the upgraded dependency miss the newest upgrade	<dependency> <groupId>group_id</groupId> <artifactId>artifact_id</artifactId> - <version>latest_version</version> + <version>order_version</version> </dependency>

dependency), or from a version number to null (removing a dependency) For example, HIVE-3256[2] upgrades library asm from 3.1 to null, i.e., removing this dependency. But the classpath for asm is still retained, which breaks the Eclipse build.

The fix for Pattern 2 is to remove the redundant or add the missing dependency in build scripts.

Pattern 3 aims at counter cases of Pattern 1. Clients sometimes make libraries under the same groupId share the same version. However, it is possible that the versions of libraries under the same organization are not synchronized. For example, SOLR-4451[3] upgraded the versions of libraries under the org.apache.httpcomponents group to 4.2.3. But httpclient 4.2.3 actually depends on the older version of httpcore.

[2] `https://issues.apache.org/jira/browse/HIVE-3256`
[3] `https://issues.apache.org/jira/browse/SOLR-4451`

The fix for Pattern 3 is to arbitrarily choose one dependency from libraries that belong to the same groupId with L, and revert its version to the the older one. If the patch does not pass, then select another one to conduct the revert. The reverting action will be repeated until a patch passes the evaluation or all dependencies belonging to the same group are traversed.

5.2 Approach of LibugFix

We propose a technique, LibugFix, to repair compilation failures by fixing build scripts From inspecting TLU-build bugs that cause compilation errors, we summarize some common patterns used to modify the build scripts (Section 5.1). All the common patterns are found from three types of build scripts: .xml files, .properties files, and .classpath files. The reason may be because the three types of files share similar semantic features. Therefore, LibugFix focuses on fixing build scripts of the three mentioned types. There are **14** TLU-build bugs whose build scripts belong to the three special types. LibugFix will conduct repair work on the 14 TLU-build bugs that cause compilation errors.

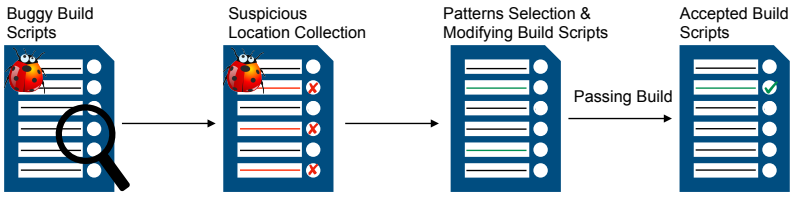

Figure 5.1: Overview of LibugFix. LibugFix first collects a list of suspicious locations, then chooses patterns to fix the build scripts. The successful compilation is used to validate the patch.

LibugFix takes three inputs to generate patches: 1) the buggy build scripts; 2) the upgraded third-party library L; 3) the version change $V < v1, v2 >$. Figure 5.1 presents the overview of LibugFix. LibugFix first collects a list of suspicious locations from the buggy build scripts. Then, it selects proper patterns to fix the build scripts. LibugFix uses the successful build as its measurement to validate generated patches.

5.2.1 Suspicious Location Collection

Source code repair tools detect suspicious locations based on test executions. However, our target bugs lie in build scripts, so we use the following rules to decide suspicious locations related to L in build scripts.

(1) All property specifications about upgraded library L.

(2) All property specifications about libraries which share the same groupId with L

A dependency specification is composed of many property specifications. In the example shown in Figure 3.2, a dependency specification includes three property specifications, which specify the groupId, artifactId, and version number of a library.

5.2.2 Pattern Selection

According to the suspicious locations collected in the previous stage, LibugFix automatically selects an available pattern to generate fixes based on rules as follows. LibugFix uses pattarns (see Table 5.1) found in RQ6a (Section 5.1).

For a given bug, if we find properties specifications related to L are not synchronized, we select Pattern 1 to fix it. If these collected properties specifications share the same version, we select Pattern 3 to fix it. If either one value of the given version change $V < v1, v2 >$ is null, we select Pattern 2 to fix it.

5.2.3 Patch Evaluation

To fix build scripts for repairing a build failure, we need a new measurement to validate the generated bug fixes, that is, we use successful compilation as our oracle. The absence of test case validation may lead to plausible patches, as the repaired program may encounter test failures.

However, the measurement is a tradeoff between fix effectiveness and efficiency. The reasons are as follows. Through our empirical study, we find TLU-build bugs are more likely to appear in large programs, i.e., a programs including multiple submodules. These large programs always have long development histories and have a large number of test cases. Running test cases once for them may even needs several days. Another reason is, for those TLU-build bugs that are detectable dynamically, most of them (6 out of 9) are runtime failures and no test cases can reveal them. Therefore, using test cases executions to validate bug fixes for TLU-build bugs is not worth the time and effort.

5.3 Fix Results (RQ6b: How effective is LibugFix in fixing build scripts?)

Table 5.2: Fix results of LibugFix.

Project	# Bug	# Fixed	Pattern 1	Pattern 2	Pattern 3
ACCUMULO	1	0	0	0	0
HADOOP	3	3	0	2	1
HBASE	1	1	1	0	0
HIVE	4	2	0	2	0
ZOOKEEPER	1	1	1	0	0
SOLR	2	2	0	1	1
GitHubProj	2	0	0	0	0
Total	14	9 (64.29%)	2 (14.29%)	5 (35.71%)	2 (14.29%)

The fix results of LibugFix answer RQ6b. For the 83 bugs that we collected, the fixes for 28 of them only involve build script modifications. As our tool conducts repair on .xml, .properties, and .classpath files, we only focus on bugs whose fixes modify build scripts of the three types. Also, we only focus a subset of TLU-build bugs that causes compilation failures. According to this rule, we choose 14 bugs from them. Table 5.2 lists the fix results of LibugFix.

```
1   <project xmlns="http://maven.apache.org/POM/4.0.0" ... >
2     ...
3     <dependencies>
4       ...
5       <dependency>
6         <groupId>com.amazonaws</groupId>
7         <artifactId>DynamoDBLocal</artifactId>
8   --    <version>${aws-java-sdk.version}</version>
9   ++    <version>1.11.86</version>
10      </dependency>
11        ...
12    </dependencies>
13    ...
14  </project>
```

Figure 5.2: Bug fix generated by LibugFix for HADOOP-14283-14589.

LibugFix successfully generates patches for 9 bugs. Specifically, among the 9 bugs, 2 bugs are fixed with pattern 1, 5 bugs are fixed with pattern 2, and 2 bugs are fixed with pattern 3.

```
  <project xmlns="http://maven.apache.org/POM/4.0.0" ... >
    ...
    <dependencies>
      ...
      <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>DynamoDBLocal</artifactId>
--      <version>${aws-java-sdk.version}</version>
++      <version>${dynamodb.local.version}</version>
      </dependency>
        ...
++    <dynamodb.local.version>1.11.86</dynamodb.local.version>
    </dependencies>
    ...
  </project>
```

Figure 5.3: Bug fix generated by clients for HADOOP-14283-14589.

Figure 5.2 gives an example of a patch generated by LibugFix for bug HADOOP-14283-14589. Clients update their AWS-SDK dependency version from 1.11.86 to 1.11.134. Before the version 1.11.134, AWS-SDK dependency always has DynamoDBLocal, so clients have DynamoDBLocal share the same version number with AWS-SDK. However, AWS-SDK 1.11.134 does not have DynamoDBLocal anymore. When clients upgrade the AWS-SDK dependency, as DynamoDBLocal is set to share the same version with AWS-SDK, the version of DynamoDBLocal is also changed by mistake, but Maven cannot find the com.amazonaws-DynamoDBLocal 1.11.134, which stops the s3guard build.

The fix generated by LibugFix reverts the version of DynamoDBLocal, which is semantically equal to the fix generated by clients.

Chapter 6

Threats to Validity

6.1 Data Collection

For collecting upgrade reasons, we examine upgrade related commits for 14 popular Java third-party libraries [51], because we cannot study all the existing third-party libraries considering the large amount of them. So the upgrade reasons that we summarize from these upgrade related commits may not reflect all reasons for other library upgrades.

Moreover, since we get 15,513 library upgrade commits that are used to identify library upgrade reasons, it is prohibitively expensive to examine all of them manually. We use the keyword search on commit messages, to extract all library upgrade related commits that possibly specify upgrade reasons. We summarize these keywords by manually inspecting upgrade related commits for the top 5 of the 14 target libraries. Then we use these keywords to process commit messages for the remaining 9 libraries and extract useful information. The keyword-based approach might miss some upgrade reasons of which the clarifications do not include keywords used by our search. Meanwhile, the summarization work is conducted by two people including the thesis author and another labmate. Hence the keywords used in this data collection may be not objective enough.

Furthermore, we find TLU bugs from: (1) 539,194 JIRA issues; (2) 1,098,764 GitHub commits. To reduce the manual work, we also utilize a keyword-based approach. We may miss some TLU bugs due to incomplete keywords.

6.2 Manual Classification

To classify library upgrade reasons, we gather different types of upgrade reasons when we look through commit messages. This work is done by the thesis author and another labmate, which may lead to bias. We will use more approaches to collect library upgrade reasons in the future, e.g., conducting a questionnaire survey, or utilizing natural language processing techniques to help process commit messages.

6.3 LibugFix

LibugFix aims to repair TLU-build bugs that cause compilation errors by fixing build scripts. We propose LibugFix based on the patterns we learned from our empirical study. For the evaluation of LibugFix, we evaluate it on the same data as what we use to summarize the fix patterns because of the insufficiency of collected TLU-build bugs, which may introduce overfitting problems. LibugFix might not show the same performance on other different TLU-build bugs. We should collect more TLU-build bugs in the future, and perform LibugFix on a larger dataset.

Chapter 7

Conclusions and Future Work

This thesis studies bugs caused by inappropriate Third-party Library Upgrades (TLUs). Bugs of this type are known little before. We first summarize the motivations for upgrading a third-party library. We also classify the types of TLU bugs. We find 41.94% of them are able to be detectable statically (compilation failures), and the rest can be detected dynamically (test failures (37.10%) and runtime failures (20.97%)). In addition, the efforts on repairing TLU bugs are investigated. We find that 16.87% of these bugs need fix efforts across multiple sources, 20.48% of these bugs can be fixed by merely modifying the source code, and 32.53% of them can be repaired by only fixing build scripts.

As current program repair tools focus on source code repair, we specifically inspect the fixes of TLU bugs which only modify the source code. We find nearly half of these bugs can be fixed by four common ways: null checking, deletion, moving the buggy statement, and replacement. We also evaluate four state-of-art program repair tools to explore whether they are effective for fixing TLU bugs, and we find that each tool can only cover a small subset of these bugs.

Furthermore, we propose an approach, LibugFix, to automatically fix build scripts. Evaluation results show that LibugFix can successfully fix 9 out of 14 bugs.

For future work, we plan to explore the following aspects.

Enlarging search space: Most of the current program repair tools use ingredients from the programs being repaired to generate patches. However, using the redundancy of the current program seems insufficient for fixing TLU bugs, because when upgrading a third-party library, new components are introduced along with the newer version of the library. So a correct fix may need new ingredients outside the current program. In the future,

we plan to find new ingredients to enlarge the search space, like new ingredients from stackoverflow or external projects.

Repairing build scripts: In this thesis, we fix build scripts using three simple patterns, which can only generate patches a limited number of TLU bugs. We plan to improve our repair tool by finding more patterns that can cover various kinds of bugs.

Repairing obsolete tests: Our study finds that 22.89% of inappropriate TLUs cause obsolete test issues. One challenge to repair obsolete tests is that we do not have oracles for it. Fixing obsolete tests is also an interesting topic in the future.

Bibliography

[1] P. Abate and R. Di Cosmo. Predicting upgrade failures using dependency analysis. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 145–150. IEEE, 2011.

[2] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[3] S. Bae, H. Cho, I. Lim, and S. Ryu. Safewapi: Web api misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 507–517. ACM, 2014.

[4] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317. ACM, 2014.

[5] 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 *ICSM*, pages 280–289, 2013.

[6] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[7] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000.

[8] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the 1996 International Conference on Software Maintenance*, page 359. IEEE Computer Society, 1996.

[9] B. Dagenais and M. P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 599–602. IEEE, 2009.

[10] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov. Reassert: a tool for repairing broken unit tests. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1010–1012. IEEE, 2011.

[11] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 207–218. ACM, 2010.

[12] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444. IEEE Computer Society, 2009.

[13] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[14] M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *2010 ACM/IEEE 32nd International Conference on Software Engineering, ICSE 2010*, pages 145–154. IEEE, 2010.

[15] J. Dietrich, K. Jezek, and P. Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 64–73. IEEE, 2014.

[16] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, volume 4067, pages 404–428. Springer, 2006.

[17] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software: Evolution and Process*, 18(2):83–107, 2006.

[18] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 15–24. ACM, 2010.

[19] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 131–142. IEEE, 2000.

[20] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang. Is this a bug or an obsolete test? In *European Conference on Object-Oriented Programming*, pages 602–628. Springer, 2013.

[21] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on Software engineering*, pages 274–283. ACM, 2005.

[22] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 39–48. IEEE, 2015.

[23] A. Hora and M. T. Valente. apiwave: Keeping track of api popularity and migration. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 321–323. IEEE, 2015.

[24] S. H. Jensen, M. Madsen, and A. Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69. ACM, 2011.

[25] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software: Evolution and Process*, 19(2):77–131, 2007.

[26] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 102–112. IEEE Press, 2017.

[27] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[28] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, pages 1–34, 2017.

[29] R. Lämmel, E. Pek, and J. Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1317–1324. ACM, 2011.

[30] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[31] 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, 2012.

[32] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[33] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[34] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, volume 2017, 2017.

[35] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[36] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, volume 51, pages 298–312. ACM, 2016.

[37] M. Martinez and M. Monperrus. Astor: A program repair library for java. In *Proceedings of ISSTA*, 2016.

[38] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 492–495. ACM, 2014.

[39] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[40] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 448–458. IEEE Press, 2015.

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

[42] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. *ECOOP 2010–Object-Oriented Programming*, pages 2–25, 2010.

[43] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *ACM Sigplan Notices*, volume 45, pages 302–321. ACM, 2010.

[44] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[45] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392. ACM, 2009.

[46] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[47] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 33. ACM, 2012.

[48] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 925–935. IEEE Press, 2012.

[49] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[50] Z. Qi, F. Long, S. Achour, and M. 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*, pages 24–36. ACM, 2015.

[51] D. Qiu, B. Li, and H. Leung. Understanding the api usage in java. *Information and Software Technology*, 73:81–100, 2016.

[52] 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. IEEE, 2012.

[53] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*, page 56. ACM, 2012.

[54] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 471–482. IEEE Press, 2015.

[55] K. Taneja, D. Dig, and T. Xie. Automated detection of api refactorings in libraries. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 377–380. ACM, 2007.

[56] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *ACM Sigplan Notices*, volume 43, pages 295–312. ACM, 2008.

[57] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294. IEEE Computer Society, 2009.

[58] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. German, and D. Poshyvanyk. License usage and changes: a large-scale study of java projects on github. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 218–228. IEEE Press, 2015.

[59] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 295–306. IEEE, 2009.

[60] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 356–366. IEEE Press, 2013.

[61] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 231–240. IEEE, 2006.

[62] J. Wloka, B. G. Ryder, and F. Tip. Junitmx-a change-aware unit testing tool. In *Proceedings of the 31st International Conference on Software Engineering*, pages 567–570. IEEE Computer Society, 2009.

[63] Z. Xing and E. Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

[64] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.

[65] J. Xuan, M. Martinez, F. DeMarco, M. Clement, 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, 2017.

[66] G. Yang, S. Khurshid, and M. Kim. Specification-based test repair using a lightweight formal method. *FM 2012: Formal Methods*, pages 455–470, 2012.

[67] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, pages 826–836. IEEE Press, 2012.

[68] Z. Zhang. xwidl: modular and deep javascript api misuses checking based on extended webidl. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 63–64. ACM, 2016.

[69] X. Zheng and M.-H. Chen. Maintaining multi-tier web applications. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 355–364. IEEE, 2007.

[70] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. *ECOOP 2009–Object-Oriented Programming*, pages 318–343, 2009.