An Empirical Investigation of Software Merging Challenges

by

Mehran Mahmoudi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Mehran Mahmoudi, 2018
Abstract

With the rise of distributed and global software development, branching has become a popular approach that facilitates collaboration between software developers. Similarly, forking, the practice of cloning an entire repository and creating an independently modified variant of it, is also common. One of the biggest challenges that developers face when employing these practices is dealing with integration problems such as merge conflicts. Conflicts occur when inconsistent changes happen to the code. Resolving these conflicts can be a cumbersome task as it requires prior knowledge about the changes in each of the versions that are going to be merged.

In this thesis, we investigate merge conflicts in different integration contexts to understand what causes them, and we identify possible directions to help developers in the integration process. We perform two empirical studies, each with a different focus.

In our first empirical study, we focus on understanding the integration challenges and improvement opportunities in independently modified variants of a repository. As a case study, we investigate the Android operating system and a community-based variant of it, LineageOS. We analyze changes made in LineageOS versus changes made in the original development of Android. By investigating the overlap of different changes, we also determine the possibility of having automated support for merging them. Our findings show that 83% of subsystems modified by LineageOS are also modified in the next release of Android. By taking the nature of overlapping changes into account, we assess
the feasibility of having automated tool support to help phone vendors with the Android update problem. Our results show that 56% of the changes in LineageOS have the potential to be safely automated.

In our second empirical study, we focus on the relationship between refactoring changes and conflicts. Previous studies have proposed techniques for facilitating conflict resolution in the presence of refactorings. However, the magnitude of the impact that refactorings have on merge conflicts has never been empirically evaluated. We study almost 3,000 well-engineered open-source Java software repositories and investigate the relation between merge conflicts and 15 popular refactoring types. Our results show that refactoring operations are involved in 22% of merge conflicts, which is remarkable taking into account that we investigated a relatively small subset of all possible refactoring types. Furthermore, certain refactoring types, such as Extract Method, tend to be more problematic for merge conflicts. Our results also suggest that conflicts that involve refactored code are usually more complex, compared to conflicts with no refactoring changes.
Preface

Chapter 4 of this thesis has been published as M. Mahmoudi, S. Nadi, “The Android Update Problem: An Empirical Study,” Proceedings of the 15th International Conference on Mining Software Repositories.

Chapter 5 of this thesis was a collaboration with Dr. Nikolaos Tsantalis from Concordia University and Dr. Sarah Nadi, my advisor. This chapter has been published as M. Mahmoudi, S. Nadi, N. Tsantalis, “Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts,” 26th IEEE International Conference on Software Analysis, Evolution and Reengineering.

This work was funded through the Natural Sciences and Engineering Research Council, NSERC.
To my mother

For always being supportive of my academic career,

even when I decided to move 10,000 kilometers away from my hometown.
Acknowledgements

First of all, I would like to thank my advisor, Dr. Sarah Nadi. Her guidance and advice was truly helpful and essential during the course of my studies. In times when I made mistakes, or I was simply lost, she always gracefully and patiently helped me learn where I had gone wrong and how I could correct my path.

I would like to thank my mother and my sister, for being supportive of me and my decisions. Moving to Canada and pursuing my Master’s degree would have been considerably more difficult, if not impossible, without their love and support.

Finally, I would like to thank my friends Amir, Benyamin, Sara, Erick, Arash, and Pouneh for all the memorable moments that we have had throughout the past two years. I was lucky to be around such amazing people.
Contents

1 Introduction 3
  1.1 Integration Challenges in Independently Modified Variants 3
    1.1.1 Motivation 3
    1.1.2 Study Overview 3
  1.2 Refactorings and Merge Conflicts 4
    1.2.1 Motivation 5
    1.2.2 Study Overview 7
  1.3 Thesis Contributions 8

2 Related Work 9
  2.1 Software Merging 9
  2.2 Software Product Lines (SPLs) 10
  2.3 Software Evolution 11
    2.3.1 API Migration 12
    2.3.2 Refactoring 12

3 Background 14
  3.1 Software Merging 14
    3.1.1 Merging in Git 14
    3.1.2 Merge Conflict 15
  3.2 Android 16
    3.2.1 Android Architecture 16
    3.2.2 Android Subsystems 18
    3.2.3 LINEAGEOS 18
  3.3 Software Evolution 18
    3.3.1 Detecting Refactoring Changes 19
    3.3.2 Detecting Other Code Changes 19

4 The Android Update Problem 21
  4.1 Methodology and Tool Chain 23
    4.1.1 Step 1: Identify All Methods 24
    4.1.2 Step 2: Populate Changesets 24
    4.1.3 Step 3: Map Changesets 27
  4.2 Study Setup 32
    4.2.1 Identifying comparison scenarios 32
    4.2.2 Downloading repositories & identifying subsystems 33
    4.2.3 Performing the analysis 33
  4.3 Results 33
    4.3.1 Frequently Modified Subsystems 33
    4.3.2 Overlapping Changes 35
    4.3.3 Overall Feasibility of Automation 38
  4.4 Discussion 41
    4.4.1 ‘Possible’ Categories 42
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Refactoring types detected by RefactoringMiner</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Mappings of changesets and the feasibility of automation in a comparison scenario</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Java subsystems in each comparison scenario</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Most changed subsystems in AN and LinOS</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Heat map of average proportion of change types across all comparison scenarios</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Stored code ranges for each refactoring type</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Statistics for merge scenarios, merge conflicts, and refactorings</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>Wilcoxon signed-rank paired test results between overall and involved refactorings. When involved refactorings are more than overall refactorings, the direction of difference is ↑, and ↓ otherwise. Statistically significant results ($p &lt; 0.05$) are shown in bold, with highlighted rows being specifically of interest.</td>
<td>68</td>
</tr>
</tbody>
</table>
List of Figures

1.1 A sample merge conflict caused by a refactoring operation . . 6

3.1 An overview of a merge scenario. Each labeled line represents
a branch, and the black-dotted commits constitute the merge
scenario . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

3.2 Android platform architecture [3]. Image used in accordance to
Creative Commons 3.0 Attribution License . . . . . . . . . . . 17

4.1 An overview of a comparison scenario in a given subsystem.
The three versions within the dashed area are those used in the
comparison scenario . . . . . . . . . . . . . . . . . . . . . . . . 23

4.2 An overview of our tool chain for identifying and comparing
changes in a comparison scenario. Each square corresponds to
a step and the oval below it is the tool used in that step . . . . . 24

4.3 Example method in AO from SystemUI subsystem that went
through different body-only changes in AN and LinOS . . . . . 39

4.4 (a) Percentage and (b) count of LinOS changed methods, cat-
gegorized by feasibility of automated merging into AN for each
comparison scenario . . . . . . . . . . . . . . . . . . . . . . . . 40

5.1 An overview of our methodology for detecting involved refac-
torings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

5.2 Distribution of conflicting region size with and without involved
refactorings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

5.3 Number of evolutionary commits per merge scenario with and
without involved refactorings . . . . . . . . . . . . . . . . . . . 64

5.4 Percentage of each involved refactoring’s type per project . . 66
Chapter 1

Introduction

Version control systems (VCSs), which keep track of the software development history, have become an essential component of modern software development. With the increase of distributed and global software development [50], [53], additional coordination tools and processes have been introduced to facilitate collaborative software development.

A common example of such practices is branching, which is particularly popular in large software systems. A branch is an instance of the source code. In this approach, developers create multiple development branches and apply their changes in parallel. Developers might decide to follow a branch-based approach for different reasons. Microsoft published a comprehensive guide [18] on different reasons teams follow a branch-based workflow. These reasons include isolating development work, bug fixes, and releases. In a survey of software practitioners, Phillips et al. [82] confirmed that branch-based development is indeed popular.

Another common practice in collaborative software development is forking. It is the practice of cloning an entire repository and creating an independently modified variant of it [85]. Some web-based source-code hosting services, such as GitHub and Bitbucket, offer systematic support for forking existing repositories. Developers working on forked repositories often need to integrate and sync up their code base with the original repository. Web-based services assist developers in the integration process by providing a graphical user interface to compare the status of both repositories. Nonetheless, developers may choose
not to use such services and perform the integration manually, especially if they have not forked the repository using these services to begin with.

While both branching and forking have several advantages such as allowing better separation of concerns and enabling parallel development [97], they still come at the cost of integration challenges [17]. Once a developer has completed the intended work in a given branch, they need to merge their changes with the rest of the team’s work. In the case of forking, developers often need to sync up with the latest changes in the main repository to make sure their clone reflects the most recent changes. We refer to any of these situations as a merge scenario.

In any given merge scenario, merge conflicts may arise, because of inconsistent changes to the code. Previous studies have shown that up to 16% of merge scenarios lead to conflicts [19]. Developers have to resolve such conflicts before proceeding, which wastes their time and distracts them from their main tasks [71]. A recent study that focused on practitioners’ perspectives on merge conflicts shows that one of the things developers struggle most with when resolving conflicts is understanding why the conflict occurred, especially in the context of more complex conflicts [70].

In this thesis, we investigate merge conflicts in different integration contexts to understand the kind of code changes that cause conflicts, and we identify possible directions to help developers in the integration process. We perform two empirical studies each with a different focus: (i) understanding the integration challenges in independently modified variants of a repository and (ii) investigating the relationship between refactoring changes and conflicts. In our first empirical study, we investigate the Android operating system which is notorious for having multiple cloned variants that often lag behind. As part of our results for that study, we find that refactorings make up the majority of changes that have the potential to be automatically integrated, and are currently causing conflicts. Therefore, we study refactoring changes in more detail and at a larger scale in our second empirical study. More specifically, we study a large number of repositories and investigate the relation between refactoring changes and merge conflicts.
1.1 Integration Challenges in Independently Modified Variants

Challenges that developers face in independently modified variants of a repository could be more comprehensive compared to those in branch-based development. This is because developers who work on forked projects often do not belong to the original repository’s development team. A good example of this situation is Google’s open-source mobile operating system (OS), Android. Using the Android Open Source Project (AOSP), phone vendors are able to access Android’s source code, clone it, and implement their own changes, such as device specifications and drivers [3]. When a new version of AOSP is released, phone vendors need to obtain the new version and re-apply their modifications to it.

1.1.1 Motivation

Due to the complexity of the migration task that phone vendors face, the majority of devices that use Android may not run on the most recent version right away. Based on data collected by Google in July 2017 [5], 27% of Android-based devices run an Android version that is at least three years old, which is especially problematic for security updates [92], [93].

The above adoption lag motivated us to study the update and integration process in Android. We propose a methodology for evaluating the challenges and difficulties that vendors face when they try to integrate their independently modified variant of Android with a new version, and we investigate automation opportunities.

1.1.2 Study Overview

To investigate automation opportunities, it is imperative to first have an understanding of the changes that vendors make versus those that Android developers make, and whether these changes overlap.

Thus, in this study, we focus on understanding the nature of the changes that occur in a large software system such as Android when compared to
an independently modified version of it. If we understand the nature of the changes on a semantic level (e.g., add method argument), we can identify current tools and techniques that can address them, or identify technology gaps that need to be filled.

Since we do not have access to proprietary vendor-specific code, we use a popular community-based variant of Android, LINEAGEOS, as a proxy for a vendor-based version of Android. For each subsystem in AOSP, we track 12 types of method-level changes in the source code using a combination of existing code-evolution analysis tools. We are interested in changes in two directions. First, between an old version of Android and its subsequent new version. Second, between the old version of Android and the independently modified LINEAGEOS version that is based on that old Android version.

We then analyze and discuss all possible combinations of change types for the two computed sets of changes. For example, a method in an old version of Android is renamed in the new Android versions, while it is moved in the corresponding LINEAGEOS versions. Based on our discussions, we determine whether automation is possible for a given combination of types of changes. We then estimate the proportion of changes that can potentially be automated.

Our results show that on average, 56% of LINEAGEOS changes have the potential to be automated when integrating AOSP’s changes. However, for 28% of the cases, developer input would be needed. The automation feasibility of the remaining 16% depends on the specifics of the change.

This study was published in Proceedings of the 15th International Conference on Mining Software Repositories [65]. We discuss this study in detail in Chapter 4.

1.2 Refactorings and Merge Conflicts

Our findings from our empirical study on Android suggest that the majority of method changes that can be automated are those that remain identical in the new version of Android and are changed in LINEAGEOS. Textual merge tools can already handle these kinds of changes and automatically merge them.
The remaining types of changes that have the potential to be automatically automated are those with refactorings. However, our findings at this point are based on only one case study. Hence, our second empirical study focuses on the relationship between refactoring changes and merge conflicts on a large scale.

Fowler et al. \cite{43} define refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”. Refactoring is used to enhance the software with regards to reusability, modularity, extensibility, maintainability, etc. \cite{72}. It is also utilized in software reengineering \cite{29}, which involves the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

An example of how a refactoring operation might be involved in a merge conflict is shown in Figure 1.1. Here, Alice moves function \texttt{foo()} from \texttt{Foo.java} to \texttt{FooHelper.java}, while Bob adds the line \texttt{x += 2;} to \texttt{foo}’s implementation in its original place in \texttt{Foo.java}. The figure shows the resulting conflict in \texttt{Foo.java}, when Bob tries to merge his code with Alice. As shown, the resulting conflict in \texttt{Foo.java} shows that the whole function is deleted in one branch, but modified in the other; the number of conflicting lines reported is also large (the size of the whole function \texttt{foo}). Given that previous research shows that developers look at the number of conflicting lines as a means of assessing how much time and effort resolving a conflict would take \cite{70}, Bob would mistakenly think that this is a complex conflict that would take him lots of time to understand and resolve, especially since he is not aware that he also needs to look at \texttt{FooHelper.java} to understand what happened. In reality, the conflict is actually simple: Alice moved the function (a refactoring operation) while Bob added an extra piece of code to it. A simple resolution would be to add the extra piece of code to the new location of the function.

1.2.1 Motivation

The example in Figure 1.1 demonstrates how refactorings may complicate the merging process. In our empirical study on Android in Chapter 4, we propose
Figure 1.1: A sample merge conflict caused by a refactoring operation
strategies for handling such conflicts. Dig et al. [33] proposed merging techniques in the presence of refactorings. Other researchers focused on improving code matching and resolution precision in software merging by considering specific types of refactorings, such as renamings [6], [61].

Researchers agree that refactorings may potentially complicate a merge scenario. However, how often does this occur in practice? Do refactorings actually result in more complex conflicts? Understanding the relationship between refactoring and merge conflicts on a large scale is important to drive researchers’ efforts in the right direction. This is why we perform a comprehensive empirical study to investigate this relationship.

1.2.2 Study Overview

As opposed to related work that looked at a small number of repositories [33] or at a couple of refactoring operations [6], [61], our second study in this thesis analyzes close to 3,000 GitHub repositories and uses a state-of-the-art refactoring detection tool. To the best of our knowledge, this is the first large-scale empirical study on the relationship between refactorings and merge conflicts.

We find that 22% of merge conflicts involve refactoring, which is remarkable taking into account that we investigated only 15 refactoring types while refactoring books describe more than 70 different types [43]. This shows that refactoring changes end up being involved in a considerable portion of merge conflicts, and suggests useful potential for refactoring-aware merging techniques.

Furthermore, we find that conflicts that involve refactorings are more complex than conflicts with no refactoring. This reaffirms the necessity of tools and techniques that can assist developers in the merging process in the presence of refactorings.

Our results also show that when adjusted for their overall frequency, refactoring types affect conflicts at different rates. Extract Method refactorings are involved in more conflicts than their typical overall frequency, while the majority of refactoring types are involved in conflicts less frequently. This is
bad news for current refactoring-aware merging techniques that rely on “un-applying” refactoring operations and calls for more sophisticated approaches.

This study was published in the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering. We discuss this study in details in Chapter 5.

1.3 Thesis Contributions

The findings of this thesis shed light on the integration challenges for independently modified variants of a system (through a case study of Android) and pave the road for refactoring-aware merging tools. Concretely, the contributions of this thesis are:

- A methodology for analyzing method-level changes between an independently modified variant of a repository and the original repository, including a breakdown of the overlap between these changes and the feasibility of automation for each category, and a discussion of the suitability of current tooling for the task.

- The first research work to discuss the Android update problem, i.e., the problem of integrating vendor-specific changes with AOSP updates. We use the Android update problem as a case study of our methodology, where we analyze the 8 latest releases of AOSP versus their community modified counterparts.

- An empirical study on almost 3,000 open-source Java repositories to investigate the role of refactoring operations in merge conflicts.

- A methodology for detecting refactorings in evolutionary changes that lead to merge conflicts.

- Open-source implementation of our methodology for both studies [45], [46], to facilitate verification and replication efforts.
Chapter 2

Related Work

In this chapter, we discuss previous work in the literature related to the topics of this thesis. We categorize related work into 3 categories: Software Merging, Software Product Lines, and Software Evolution.

2.1 Software Merging

While branch-based development and forking are common practices in software engineering, integration challenges for software merging remain a drawback. There are multiple studies that propose new merging techniques to reduce the manual integration labor as well as to decrease the likelihood of merge conflicts. According to the seminal survey by Mens \cite{mens71}, software merging tools can be categorized by how they represent software artifacts.

*Text-based* merge tools are language-independent and consider software artifacts as text-files \cite{14, 64}. Because of their line-based approach, these tools cannot handle simultaneous changes to the same lines. The conflicts we study in this paper, as reported by Git, are based on text-based merge tools.

*Syntactic* merge tools are more advanced since they take into account the syntax of software artifacts \cite{21, 78}. These tools can ignore unimportant conflicts such as code comments or line breaks. While these tools can ensure that the merged program is syntactically correct, they cannot prevent semantic conflicts.

*Semantic-based* merge tools overcome these type of conflicts by employing ASTs, dependency graphs, program slicing, and denotational semantics \cite{7},
Operation-based merge tools, which are a flavor of semantic-based tools, consider changes between versions as operations. Nishimura et al. \cite{77} proposed a tool that assists developers with merge conflicts. Their approach reduces the burden of manual inspection for developers by replaying fine-grained code changes related to conflicting class members. Dig et al. \cite{33} proposed MOLHADORef, a software configuration management system that is aware of refactoring operations. MOLHADORef merges two software revisions by inverting the refactoring operations, performing a textual merge, and replaying the refactoring operations. However, they did not empirically study how often refactorings cause conflicts, and how effective their approach is on a large scale.

In addition to improving software merging algorithms themselves, some researchers proposed continuously running or merging developer changes in the background to warn developers about potential conflicts before they actually occur \cite{19, 31, 49}. Furthermore, some researchers performed empirical studies to predict merge conflicts \cite{1, 62} or to understand practitioners’ views on conflicts \cite{70}.

Finally, there are a number studies that investigate collaborative software development in the context of forked repositories and pull-request model \cite{47, 48, 102}. These studies mainly focus on whether a pull-request would be integrated or rejected by a reviewer from a code change perspective and do not take into account merge conflicts.

### 2.2 Software Product Lines (SPLs)

The idea of consolidating independently evolved versions of a software system, or forking, is also related to the SPL domain. Various studies and tools have been proposed to identify commonalities and variability between source code of multiple software versions and consolidate them in an SPL (e.g., \cite{35, 38, 40, 87}). As opposed to traditional merging, the goal with SPLs is to integrate the variants but keep them configurable such that it is possible to choose any
of the existing behavior.

There are a number of studies on forked repositories that investigate different aspects. For example, Ray et al. [85] investigated the sustainability of forked repositories by studying 18 years of the BSD product family history. They find that forked projects require significant maintenance effort. Moreover, Businge et al. [22] studied reuse practices to identify families of cloned Android applications and find that code propagation is not common among them. However, to the best of our knowledge, there is no study that investigates the integration challenges in cloned variants of the Android OS.

We would like to note that there is existing work that studies techniques and requirements for providing a dynamically updatable operating system [10], [11]. Obviously, if Google decides to change the Android architecture to support such dynamic updates on all levels of the OS, then the update problem that we discuss in Chapter 4 would be solved. However, we look at the current state of the Android OS and address the Android update problem from the perspective of phone vendors, not from the perspective of the OS owners.

2.3 Software Evolution

In our first empirical study in Chapter 4 we investigate the integration challenges in an independently variant of the Android OS. Understanding how a software system evolves over time, especially in the context of collaborative software development, can provide more details about the integration challenges that developers face.

A lot of work focused on identifying types of changes that occur during software evolution [20], [56], with varying goals and underlying techniques. Some of the most notable goals that previous studies followed include finding factors for successful software reuse [89], validating hypotheses for successful open-source development [34], predicting future changes [23], [101], etc. However, to the best of our knowledge, there is no software evolution study that discusses integration challenges in cloned variants of a software system by comparing semantic changes between cloned variants and the original repository.
Our empirical study in Chapter 4 relies on several existing tools and techniques in software evolution, such as CHANGEDISTILLER [41], SOURCERC [88], and REFACTORINGMINER [94] to get a comprehensive set of method-level changes. Details about the tools we use are given in Chapter 3.

We now discuss API migration and refactoring, two topics in software evolution that are related to this thesis, in more details.

### 2.3.1 API Migration

The techniques used to identify changes during software evolution have often been used as a basis for updating client projects if a library API they use changes. For example, Xing and Stroulia [100] develop DIFF-CATCHUP based on UMLDIFF to automatically recognize API changes and propose plausible replacements to obsolete APIs. Similarly, Dagenais and Robillard [28] develop SEMDIFF, which tracks and analyzes the evolution of a framework to infer high-level changes and recommend adaptive changes to client applications.

Previous work showed that API migration is a prevalent problem in Android, where app developers do not generally keep up with the pace of API updates and often use outdated APIs [69]. Linares-Vásquez et al. [63] also study Android’s public APIs and find out that using unstable and fault-prone APIs negatively impact the success of Android apps. There are similar additional studies on Android public APIs [69]. However, our first empirical study in Chapter 4 does not focus only on updates to public APIs used by clients, but rather involves analyzing the evolution of independently maintained versions of a piece of software.

### 2.3.2 Refactoring

In our second empirical study in Chapter 5, we investigate refactoring operations and merge conflicts. Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [43]. Researchers have used refactoring detection tools to study how software evolves in the presence of refactorings. For example, how refactorings impact bugs [12], software quality [13],
or regression testing [84]. Other researchers studied why and how refactorings happen [59], [75], [79]. However, to the best of our knowledge, there is no study that investigates the relationship between refactorings and merge conflicts.

There are a number of refactoring detection algorithms and tools in the literature [30], [32], [42], [44], [60], [76], [83]. However, most of these tools have low precision and/or recall, need a similarity threshold to determine if two parts of the code are related, and require two fully-built versions of the software as an input in order to utilize type-binding information from the compiler. Aiming to mitigate such problems, Tsantalis et al. [94] recently proposed a tool, named RefactoringMiner, which we use in this work. RefactoringMiner does not require predefined similarity thresholds, operates at both commit level and file level, and achieves a high accuracy rate (a precision of 98% and recall of 87%).
Chapter 3

Background

In this chapter we explain some background information related to this thesis. We introduce terms and definitions we use in the thesis, tools we use in our work, as well as systems we analyze.

3.1 Software Merging

Using multiple branches in a source control system is a common practice in software development that serves a variety of purposes [9], [97]. At one point, developers need to integrate the changes from the different branches, and this is done by merging the corresponding branches. Since we focus on repositories that are maintained with Git in Chapter 5, we now explain merging in Git in more details.

3.1.1 Merging in Git

Almost all merge tools that are currently available, including the one utilized by Git, employ three-way merging techniques [71]. In three-way merging, two versions of a software artifact are merged by making use of an additional third version, which is often called the base version.

Figure 3.1 illustrates a typical merge scenario. When merging two branches, Git attempts to merge the most recent commit in each branch. We refer to these commits as merge parents. As a base version in a merge scenario, Git uses the most recent commit that both merge parents can be derived from, referred to as the common ancestor. The result of the merge is stored in a
Figure 3.1: An overview of a merge scenario. Each labeled line represents a branch, and the black-dotted commits constitute the merge scenario.

merge commit. A merge commit can be identified from the Git history, since it has two or more parents (namely $P1$ and $P2$ in the example of Figure 3.1), unlike typical commits that have one parent.

### 3.1.2 Merge Conflict

Based on the nature of the merge scenario, a textual three-way merge tool, such as the one used by Git, might not be able to automatically merge the two versions of a file. For a given conflicting merge scenario, Git can report conflicts across multiple files. Git categorizes conflicts into 6 types:

- **add/add**: When both merge parents add a new file with same name, but with different contents.
- **content**: When both parents apply different changes to the same file, in the same location.
- **modify/delete**: When $P1$ modifies a file, while $P2$ deletes it.
- **rename/add**: When $P1$ renames a file, and $P2$ adds a new file with the same name.
- **rename/delete**: When $P1$ renames a file, and $P2$ deletes it.
- **rename/rename**: When both parents rename a file to different names.
The first two types are at the content level, while the other four are at the file level. Content level conflicts could be caused by more than one location in the conflicting file. Git reports these conflicting locations by annotating them with <<<, ===, and >>> markers as shown in Figure 1.1. We call each of these annotated locations a conflicting region.

3.2 Android

In our first empirical study in Chapter 4 we study the Android OS. As background, we now introduce Android’s architecture, its subsystems, and the community-based variant of it we used, LineageOS.

3.2.1 Android Architecture

As shown in Figure 3.2 the Android software stack architecture consists of several layers [3]. Each layer has a different functionality and is written in a specific language. The lowest layer is the Linux Kernel level. Phone vendors are expected to include device drivers for their hardware in this layer. On top of the Linux Kernel, there is the Hardware Abstraction Layer (HAL). This layer defines a standard interface for hardware vendors to implement and allows Android to be agnostic to lower-level driver implementations. Phone vendors are responsible for interaction between the HAL implementation and their device drivers in the Kernel layer.

While development in HAL and the Linux Kernel is done in C/C++, higher-level layers such as the System Apps and Java API Framework use Java. Phone vendors may apply changes to these layers for various reasons, such as adding new functionality or building their ecosystem’s look and feel.

The Android source code is maintained in multiple repositories that collectively build up the source-tree. A list of these repositories is maintained in a repository called Android Platform Manifest [4].
Figure 3.2: Android platform architecture. Image used in accordance to Creative Commons 3.0 Attribution License
3.2.2 Android Subsystems

Since we need to understand which parts of the Android OS are modified, it would be useful to divide the Android source code into different subsystems. In application development for Android, each folder that contains an Android-Manifest.xml file is compiled and built into a .apk file, which is later installed on the phone. Conveniently, all parts of Android that are implemented in Java are considered as apps, meaning that they have an AndroidManifest.xml file in their source and are packaged into .apk files. Considering this fact, we decided to define the notion of an Android Java subsystem as a folder that contains an AndroidManifest.xml file.

3.2.3 LineageOS

Because phone vendors do not make the source code of their customized Android OS version publicly available, we need a proxy for a vendor-specific Android variant. There are a number of community-based Android OS variants available. Since most of them are open source, it is possible to use them as a proxy for a modified vendor-specific version of Android for our research purposes. LineageOS \cite{26,52} is a popular alternative operating system for devices running Android. It offers features and options that are not a part of the Android OS distributed by phone vendors. These features include native Android user interface, CPU overclocking and performance enhancement, root access, more customization options over various parts of the user interface, etc. In 2015, LineageOS had a large user base \cite{52} and a community of more than 1,000 developers. It is actually a continuation of CyanogenMod, a project that was discontinued in December 2016 and that continued under the new name, LineageOS.

3.3 Software Evolution

In this section, we introduce the different tools that we use in later chapters to track source code changes between two versions of a software.
<table>
<thead>
<tr>
<th>Code Element</th>
<th>Refactoring Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>CHANGE PACKAGE</td>
</tr>
<tr>
<td>Type</td>
<td>EXTRACT SUPERCLASS/INTERFACE</td>
</tr>
<tr>
<td></td>
<td>MOVE CLASS, RENAME CLASS</td>
</tr>
<tr>
<td>Method</td>
<td>EXTRACT METHOD, EXTRACT &amp; MOVE METHOD</td>
</tr>
<tr>
<td></td>
<td>INLINE METHOD, PULL UP METHOD</td>
</tr>
<tr>
<td></td>
<td>PUSH DOWN METHOD, RENAME METHOD</td>
</tr>
<tr>
<td></td>
<td>MOVE METHOD</td>
</tr>
<tr>
<td>Field</td>
<td>PULL UP FIELD, PUSH DOWN FIELD, MOVE FIELD</td>
</tr>
</tbody>
</table>

Table 3.1: Refactoring types detected by RefactoringMiner

3.3.1 Detecting Refactoring Changes

Tsantalis et al. [94] recently proposed a refactoring detection tool, named RefactoringMiner, which we use in this work. Table 3.1 shows the 15 refactoring types that RefactoringMiner is able to detect in different granularity levels. It has two modes of operation: (1) comparing two given files or folders to detect refactorings in them or (2) detecting refactorings in a given Git commit. It has a precision of 98% and a recall of 87%.

For our first study in Chapter 4, we use RefactoringMiner in file mode. For our second study in Chapter 5, we use it in commit mode.

3.3.2 Detecting Other Code Changes

Code change detection tools assist researchers in software evolution analysis by identifying source code changes over several versions of a program.

Spoon [81] is an open-source tool for analyzing Java source code. It creates an Abstract Syntax Tree (AST) for a given Java project. An AST is a tree representation of the abstract syntactic structure of a source code. We use Spoon for identifying all methods in a Java project and generating unique signatures for them.

SourcererCC [88] is a token-based clone detection tool with a high reported rate of 90% recall (100% for certain benchmarks) and 83% precision. It is designed for detecting both exact and near-miss clones in large-scale
projects. We use SOURCERERCC for identifying identical methods between two versions of a source code.

CHANGEDISTILLER [41] is a tool for extracting source code changes based on AST differencing. The reported precision and recall rates of CHANGEDISTILLER are 78% and 98%, respectively. We use CHANGEDISTILLER to find changes to a method’s arguments between two versions of the source code.
Chapter 4

The Android Update Problem

In this chapter, we present our first empirical study on integration challenges in Android.

Google’s open-source mobile operating system (OS), Android, is used by the majority of phone vendors [67] and currently has approximately 80% of the market share of smart phones around the world [96]. Using the Android Open Source Project (AOSP), phone vendors are able to access Android’s source code, and can implement their own device specifications and drivers [3]. Since Android is open source, phone vendors can add their own enhancements, including new hardware capabilities and new software features.

When a new version of AOSP is released, phone vendors need to obtain the new version and re-apply their modifications to it. Due to the complexity of this task, the majority of devices that use Android may not run on the most recent version right away. Based on data collected by Google in July 2017 [5], 27% of Android-based devices run an Android version that is at least three years old, which is especially problematic for security updates [92], [93].

The process of re-applying changes from an independently modified version of Android to a newer version of Android is a time-consuming and manually intensive task. While developers can use a mix of Application Programming Interface (API) migration and software merging tools to help them with the process, we are not aware of any single off-the-shelf tool that can be used to automatically accomplish this merging task. However, before attempting to automate this task, we first need to understand the changes that vendors
make versus those that Android developers make, and whether these changes overlap.

In the study presented in this chapter, we focus on understanding the nature of the changes that occur in a large software system such as Android when compared to an independently modified version of it. If we understand the nature of the changes on a semantic level (e.g., add method argument), we can identify current tools and techniques that can address them, or identify technology gaps that need to be filled. We focus on the Java parts of AOSP since Google recently announced the introduction of Project Treble \[2], \[66\] which allows easier update of the hardware-specific parts implemented in C through a new architecture of these layers. However, this does not solve the problem of vendor-specific Java changes in AOSP itself.

Since we do not currently have access to proprietary vendor-specific code, we use a popular community-based variant of Android, LINEAGEOS, as a proxy for a vendor-based version of Android. For each subsystem in AOSP, we track the method-level changes in the source code using a combination of existing code-evolution analysis tools: SOURCERERCC \[88\], CHANGEDISTRIBUTTER \[41\], and REFACTORMINGMINER \[94\]. We are interested in changes in two directions. First, between an old version of Android and its subsequent new version. Second, between the old version of Android and the independently modified LINEAGEOS version that is based on that old Android version. We first analyze the types of changes that have occurred in both directions. We then analyze the intersection of the two computed sets of changes to estimate the proportion of changes that can potentially be automated. Specifically we answer the following research questions:

**S1-RQ1** Which parts of Android are frequently modified in AOSP vs. LINEAGEOS? Understanding the commonalities between the subsystems that are modified in AOSP and LINEAGEOS sheds light on the extent of the Android update problem.

**S1-RQ2** What are the overlapping types of changes between AOSP and LINEAGEOS modifications? Knowing what subsystems are commonly
Figure 4.1: An overview of a comparison scenario in a given subsystem. The three versions within the dashed area are those used in the comparison scenario.

...co-modified in AOSP and LineageOS, while helpful, is not enough per se. In this RQ, we dive deeper and explore what types of method changes often happen in AOSP vs. LineageOS.

S1-RQ3 How feasible is it to automatically re-apply LineageOS changes to AOSP? Assessing the feasibility of automatic integration of independent modifications with AOSP provides useful insight for phone vendors, as well as developers of other similar forked software projects.

4.1 Methodology and Tool Chain

Figure 4.1 shows the relation between the evolution of LineageOS with respect to AOSP. A new LineageOS version is created based on a given Android version, Android Old (AO) in this case. Developers then evolve this LineageOS version to add their own customization to the original Android version it is based on. The final state of these customizations would be in the latest commit of LineageOS (LinOS). Simultaneously, Android developers are modifying Android to eventually release a new version, Android New (AN). We call the group of three versions required to compare Android changes to the LineageOS changes (i.e., AO, AN, and LinOS) a comparison scenario.
These three versions are surrounded by the dashed box in Figure 4.1. For each comparison scenario, we identify all subsystems that belong to AO and track their evolution through AOSP and LineageOS changes. We focus on changes at the method level, because the method-level provides a concrete and self-contained granularity level that we can reason about, and it has been used in many software evolution studies [56].

Figure 4.2 shows the steps we follow in our tool chain in order to analyze a comparison scenario. It consists of three main steps that are applied to each subsystem in the comparison scenario. We now explain each of these steps. Our tool chain, which is a combination of pertaining third-party tools and some custom tooling, is implemented in Java and Python and is available online [45].

4.1.1 Step 1: Identify All Methods

We use Spoon [81] to generate the Abstract Syntax Trees (ASTs) of all Java files in the three versions of a subsystem. Using the ASTs, we extract a signature for each method that uniquely identifies it. This gives us a set of all methods in each of the three versions of a subsystem.

4.1.2 Step 2: Populate Changesets

We call the set of method changes between two given versions of a subsystem a changeset. In this step, we want to populate two changesets, one between AO and AN and one between AO and LinOS. Our tooling receives the AO, AN, and LinOS versions of a subsystem as input and populates these two
changesets. We now describe the type of changes a method can undergo and the tooling we use to detect the change. When discussing the changes, we break down methods into signature and body. A method signature includes the package and class it is located in, its name, arguments, and return type. The method body is the implementation of the method. Changes to these two entities are not mutually exclusive, however, meaning that a method could be changed in both ways.

**Identical Methods**

Identical Methods are those that do not have any changes in neither their signature nor body. To identify identical methods, we use SourcererCC with 100% similarity threshold to look for exact clone pairs on the function level.

**Refactoring Changes**

Generally speaking, a refactoring change is a change that does not semantically alter the source code. In refactoring changes, the method signature is changed, although the body could be the same, depending on the type of refactoring. We consider the most common types of refactorings:

- **Method Move**: Class or package name of a method is changed.
- **Method Rename**: A method’s name is changed.
- **Method Inline**: A method is removed and its body is moved to the place it was originally called from.
- **Method Extraction**: Part of a method ($m_1$) is moved to a newly created method ($m_2$), and $m_2$ is called from $m_1$.
- **Argument Rename**: A method argument’s name is changed.
- **Argument Reorder**: The order of arguments is changed.
We use RefactoringMiner\(^1\) to detect refactoring changes at the method level between different versions of a subsystem. At the time of this study, RefactoringMiner could detect the first four refactorings in the above list, but could not detect argument rename or reorder. To detect these two types of refactorings, we use ChangeDistiller \(^[41]\), which extracts source code changes based on tree differencing.

**Argument Changes**

There are other types of argument changes that would not be considered as refactoring changes. These include adding a new argument or deleting one, as well as changing an argument’s type. We use ChangeDistiller \(^[41]\) to detect these three types of argument changes and add them to the respective changesets under the argument change category.

**Body-only Changes**

A method could have changed between two versions of a subsystem, but does not belong to any of the above categories. We already covered changes that can only affect a method’s signature (method move, method rename, argument rename, and argument reorder) or both its signature and body (method inline, method extraction, and argument changes). Body-only changes include methods that have the exact same signature between two versions of their subsystem, but have modified bodies.

For each of the two changesets, we implement our own tooling to search for methods in the AO version of the subsystem that have not been paired with a method in the related version. Between these unmatched methods, we look for pairs with matching signatures. Such pairs of methods will definitely have different bodies; otherwise, they would be identical and would have been discovered by SOURCERERCC when identifying identical methods. We add such methods to the body-only change category.

\(^1\)https://github.com/tsantalis/RefactoringMiner, used as of commit 85dba96
Unmatched Methods

Given two versions of a system, a method that exists in the old version might have undergone large changes that prevent existing tools from matching it with the corresponding method in the new version. The method could have also been simply deleted. We use a simplified heuristic where if we exhaust our search for a match for a method by going through all the above change types, we categorize this method as unmatched.

4.1.3 Step 3: Map Changesets

After acquiring the two changesets $AO$ to $AN$ and $AO$ to $LinOS$, we develop custom tooling that maps method changes that have the same signature in $AO$ from one changeset to the other.

This mapping allows us to understand the extent and nature of overlapping changes. Table 4.1 shows the potential overlapping changes to the same method, by considering all the possible combinations of changes. Column and row labels represent types of changes in each changeset.

The next step is to understand if integrating each category of overlapping change can be automated. To determine this, we consider the nature of the changes and categorize each cell based on the possibility of automatically integrating both changes without needing input from the developer. Green cells indicate that automation is possible. This means that it is possible to develop a tool chain that utilizes a combination of state-of-the-art tools and techniques in order to automatically merge these change types, without needing any input from the developer in order to combine the simultaneous changes from both sides. Red cells indicate cases where complete automation is not possible, because developer input is needed. In other words, for part of the code changes, a tool needs to pick one side or another and such a decision should be left up to the developer. Yellow cells indicate potential problems: depending on the details of the change, automation may or may not be possible. We have not included the identical change type between $AO$ and $LinOS$ in the table, because there are no changes to apply on $AN$ for those methods.
Table 4.1: Mappings of changesets and the feasibility of automation in a comparison scenario

<table>
<thead>
<tr>
<th>Method Move</th>
<th>Method Rename</th>
<th>Method Inline</th>
<th>Method Extract</th>
<th>Argument Rename</th>
<th>Argument Reorder</th>
<th>Argument Add</th>
<th>Argument Remove</th>
<th>Argument Type Change</th>
<th>Body-only</th>
<th>Unmatched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Might Be Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
</tbody>
</table>

28
We start by discussing our reasoning across the first row, last row, and last column of Table 4.1 as they have the same color across. If a method in AN is identical to AO, then we can safely automatically apply any change from LinOS. This is why the whole first row is green, with the exception of the last cell. This leads us to discussing the last row and last column. If a given method is unmatched in either changesets, then we conservatively assume that this is an integration that cannot be automated, and mark the cell as red. The simplest example is if the method gets deleted in one changeset but not the other. However, there may be more complicated types of changes that our tooling could not map. Therefore, we choose the more conservative assumption that these changes cannot be safely automated. Note that with the exception of the cell in the bottom right corner, all cells across the diagonal are marked as yellow, since if the same type of change is applied in both change sets, the feasibility of automation depends on the particular case. For example, if both change sets include an argument or method rename, then the integration of the changes can be automated if both change sets used the same destination name. On the other hand, if the destination names are different, then developer intervention is needed to decide which name to choose. We now discuss our reasoning for the color choice of the remaining cells, column by column, without repeating the reasoning for the first row, last row, last column, and diagonal in the table.

**AO → LinOS Method Move.** This column looks at cases where the method was moved in LinOS. If the method was inlined in AN, then it no longer exists and thus cannot be automatically moved, resulting in a red cell. If part of the method was extracted in AN, then a tool cannot automatically decide whether it should create the extracted method in the same location as that in AN or if the extracted method should also be moved to the new location in LinOS. Because this requires the developer’s decision, we color this cell red. If the arguments or body of the corresponding method in AN are changed, the move change from LinOS could still be easily applied in AN since the move operation does not interfere with any of these types of changes.

**AO → LinOS Method Rename.** This change is very similar to Method
Move. Thus, the coloring of most cells follows the same reasoning. The only exception is Method Extract in AN. If a method was extracted in AN and was renamed in LinOS, a tool will not face the same question about where should it move the extracted part. The method will be extracted in the same location as AN and then renamed similar to LinOS; thus this cell is colored green. When automating the integration of this change to AN, all calls to the method in AN should also be updated.

**AO → LinOS Method Inline.** If a method is inlined in LinOS, it means that its body was copied to all its call sites that existed in AO, and the method itself has been removed. In this change, the implementation of the method is not altered, therefore, if there has been a body-only change in AN, we can still apply this change to the inlined code. However, any other type of change in AN that alters the method’s signature cannot be easily automatically integrated. For example, assume that some argument got removed from the method’s signature, applying this change to the inlined body of the method may not be that straightforward since it would require removing parts of the code in which the old method got inlined, an operation that is best left to the developer’s judgment. The same logic applies to the methods that were inlined in AN, hence the (AO → AN Method Inline) row is also mostly red.

**AO → LinOS Method Extract.** This change occurs when part of the method’s implementation in AO is moved to a newly created method in LinOS. If the method is moved in AN, a tool would not be able to automatically determine where to create the extracted method and the choice depends on the developer’s decision. Since method extract affects a method’s body, a tool would not be able to automatically apply the change in cases where AN also altered the body (those marked in red). All the other change categories do not alter a method’s body, and thus, can be automatically integrated.

**AO → LinOS Argument Rename.** Argument renames can be automatically reapplied to AN in most cases. The only exceptions are when an argument is added or removed in AN, since the changed argument could potentially have the same name of the renamed argument in LinOS. We mark these cases as yellow.
**AO → LinOS Argument Reorder.** This change type is similar to the previous argument rename. The only difference is when a new argument is added or removed in AN. In this case, a tool cannot automatically determine how to reorder the arguments in the presence of a new argument or absence of a previously present argument. We mark both these cases as red.

**AO → LinOS Argument Add and Remove.** We discuss both these columns together since their reasoning is similar. When a new argument is added or an old one is deleted, a method’s body is often altered as well to reflect the change. Therefore, if the method body is also modified in AN through any type of change, it is not possible to automatically reapply the changes from both change sets. If an argument is renamed in AN, it would be a problem if the same argument is removed in LinOS. We will face a similar problem if the new argument name in AN is the same as the new argument added in LinOS. Hence, we mark both these cases as yellow.

**AO → LinOS Argument Type Change.** We assume that when an argument type change happens in LinOS, the body is also changed. Argument type changes can be automatically reapplied for changes in AN that only affect the method’s signature, i.e. method rename, method move, and argument reorder. Although argument rename in AN mostly likely changes the method body, a tool can still automatically reapply the argument type change in LinOS, because argument rename is merely a refactoring change than can be reapplied even if other parts of the body are changed. Other change types in AN involve non-trivial changes in the method’s body that are likely to cause conflicts with argument type change in LinOS, so they are colored red.

**AO → LinOS Body-only.** The reasoning for this change type is very similar to the previous case, since we assume an argument type change also means a change in the method’s body. The only difference is for the case where the method is inlined in AN. The body-only change from LinOS can be re-applied by simply removing the old inlined version of the method and re-performing the inline operation with the new body from LinOS.
### 4.2 Study Setup

In this section, we explain the setup we use for running our methodology from Section 4.1 on AOSP and LineageOS.

#### 4.2.1 Identifying comparison scenarios

We consider the eight latest LineageOS versions in our study. To identify a comparison scenario, we need the AOSP version each LineageOS version is based on and the subsequent version of AOSP that it will need to be updated to. Each version of LineageOS is based on a corresponding version of AOSP, but it is not necessarily based on the first release of that AOSP version. LineageOS does not modify all of the repositories used in AOSP. Since these repositories are still required to build the source tree, LineageOS fetches them from the AOSP servers. Fortunately, the exact AOSP version and release number for those unmodified repositories is mentioned in the manifest file for LineageOS that contains the list of required repositories. We assume that the repositories that LineageOS modifies are also based on this documented release number, because it only makes sense for LineageOS developers to include the unmodified AOSP repositories from the same version that modified repositories were based on; otherwise, incompatibilities may occur.

<table>
<thead>
<tr>
<th>Comparison scenario</th>
<th>AO</th>
<th>AN</th>
<th>LinOS</th>
<th># of subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS1</td>
<td>4.2.2</td>
<td>r1</td>
<td>cm-10.1</td>
<td>208</td>
</tr>
<tr>
<td>CS2</td>
<td>4.3.1</td>
<td>r1</td>
<td>cm-10.2</td>
<td>219</td>
</tr>
<tr>
<td>CS3</td>
<td>4.4.4</td>
<td>r2</td>
<td>cm-11.0</td>
<td>275</td>
</tr>
<tr>
<td>CS4</td>
<td>5.0.2</td>
<td>r1</td>
<td>cm-12.0</td>
<td>346</td>
</tr>
<tr>
<td>CS5</td>
<td>5.1.1</td>
<td>r37</td>
<td>cm-12.1</td>
<td>355</td>
</tr>
<tr>
<td>CS6</td>
<td>6.0.1</td>
<td>r81</td>
<td>cm-13.0</td>
<td>381</td>
</tr>
<tr>
<td>CS7</td>
<td>7.0.0</td>
<td>r14</td>
<td>cm-14.0</td>
<td>392</td>
</tr>
<tr>
<td>CS8</td>
<td>7.1.2</td>
<td>r36</td>
<td>cm-14.1</td>
<td>398</td>
</tr>
</tbody>
</table>

Table 4.2: Java subsystems in each comparison scenario
4.2.2 Downloading repositories & identifying subsystems

We are only interested in the mutual repositories between each version of LINEAGEOS and its corresponding AOSP version, since there is no point in studying repositories in LINEAGEOS that do not have a corresponding repository in AOSP and vice versa. After processing the manifest file that contains the list of required repositories for each version of LINEAGEOS, our tooling automatically downloads the mutual repositories between LINEAGEOS and AOSP. It recursively searches all sub-folders for AndroidManifest.xml, and considers such folders as subsystems. Table 4.2 shows the number of Java subsystems we analyzed for each comparison scenario.

4.2.3 Performing the analysis

For each comparison scenario, we apply our methodology from Section 4.1. For each subsystem in every comparison scenario, we produce an output file that consists of the number of methods in each of the three versions of a given comparison scenario (AO, AN and LinOS), as well as the number of methods in each overlapping category of changes, similar to Table 4.1.

4.3 Results

We now answer our three research questions.

4.3.1 Frequently Modified Subsystems

In order to better understand the kind of changes AOSP or LINEAGEOS undergo, our first step, reflected in S1-RQ1, is to look at the frequently modified subsystems. Table 4.3 shows the number of changed subsystems by LinOS and AN in each comparison scenario. Column 2 shows the number of changed subsystems in AN, column 3 shows the number of changed subsystems in LinOS, and column 4 shows the number of mutually changed subsystems between AN and LinOS as well as the percentage with respect to LinOS.

Table 4.3 shows that the vast majority of subsystems changed by LinOS are also changed by AN. On average across the comparison scenarios, approx-
Comparison | Number of changed subsystems in AN | Number of changed subsystems in LinOS | Number of mutually changed subsystems
--- | --- | --- | ---
CS1 | 51 | 23 | 20 (86.96% of LinOS)
CS2 | 49 | 26 | 22 (84.62% of LinOS)
CS3 | 74 | 34 | 31 (91.18% of LinOS)
CS4 | 79 | 36 | 27 (75.0% of LinOS)
CS5 | 88 | 40 | 36 (90.0% of LinOS)
CS6 | 102 | 61 | 54 (88.52% of LinOS)
CS7 | 74 | 34 | 23 (67.65% of LinOS)
CS8 | 84 | 33 | 26 (78.79% of LinOS)
Average | 75 | 36 | 30 (83.33% of LinOS)

Table 4.3: Most changed subsystems in AN and LinOS

approximately 83% of the subsystems changed by LinOS are also changed by AN. This high number illustrates the extent of the Android update problem. However, the fact that a subsystem has been modified in LinOS and AN does not tell us anything about the nature or extent of these changes. To better understand the changes that subsystems undergo, we extract the five most changed subsystems in LinOS and AN for each comparison scenario. We find that there are at least two mutually changed subsystems between AN and LinOS in all comparison scenarios. This suggests that AOSP and LINEAGEOS developers often apply a large number of changes to the same Android components. To better understand why developers need to change these subsystems, we further investigate two subsystems that seem to be commonly modified by both AOSP and LINEAGEOS developers, SystemUI and Settings.

**SystemUI**

This subsystem is responsible for core user visible components that are generally accessible from any app, e.g., the status bar and volume control sliders. The status bar can be expanded by dragging it down from the top of the screen. It provides useful features such as the list of notifications and quick settings. Since it is frequently used by Android users, it makes sense that AOSP developers frequently make changes to it. Many of LINEAGEOS-specific features are implemented in this subsystem, e.g., customizable quick settings. In Section 4.3.2, we discuss a method change sample from this subsystem which demonstrates how LINEAGEOS modifies the code to implement new features.
**Settings**

The Settings subsystem is the interface that lets users reconfigure different settings in their Android device. There are several options available in Settings, often grouped into categories such as Connectivity or Display. AOSP developers often change this subsystem to amend new features and improve the user interface of existing items within each category. Settings in LineageOS includes all options from AOSP, plus some additional LineageOS-specific items that allow users to further customize their device. For example, there is an option under the Battery category in LineageOS for profiling CPU usage. Users can change this option to determine the workload on the CPU and subsequently its battery usage.

**Answer to S1-RQ1:** On average, 83% of the subsystems modified by LinOS are also modified by AN. Settings and SystemUI are the two most commonly co-modified subsystems.

### 4.3.2 Overlapping Changes

In S1-RQ2, we further study the types of changes that happen in mutually changed subsystems to better understand the nature of changes that are applied by AOSP developers, versus LineageOS developers. To analyze this, we calculate the number of method changes and their types among all subsystems for each comparison scenario. Table 4.4 shows the average proportion of each change type across all comparison scenarios. The rows and columns, as well as the text colors, correspond to those in Table 4.1, making each cell correspond to an overlap cell in Table 4.1. The number in each cell indicates the percentage of the corresponding change overlap with respect to the total number of changes in LinOS, as an average of all comparison scenarios. Let us take the AO →LinOS:Body-only and AO →AN:Body-only cell as an example. The number in the cell indicates that on average, 16.09% of all methods that are changed in LinOS had body-only changes in both LinOS and AN. For easier visualization, the table is drawn as a heat map, where the grayscale intensity of the cell background corresponds to the percentage value in the cell.
Table 4.4: Heat map of average proportion of change types across all comparison scenarios

<table>
<thead>
<tr>
<th>AO</th>
<th>Body-only</th>
<th>Argument Type Change</th>
<th>Argument Rename</th>
<th>Argument Add</th>
<th>Argument Remove</th>
<th>Argument Reorder</th>
<th>Argument Type Change</th>
<th>Method Move</th>
<th>Method Rename</th>
<th>Method Inline</th>
<th>Method Extract</th>
<th>Changeset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmatched</td>
<td>0.07%</td>
<td>0.01%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Body-only</td>
<td>0.07%</td>
<td>0.01%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Argument Type Change</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Argument Rename</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Argument Add</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Argument Remove</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Argument Reorder</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Argument Type Change</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Method Move</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Method Rename</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Method Inline</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Method Extract</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Changeset

mutations

AO

→

LinOS

→

AN

Identical

0.28%

0.4%

0.06%

1.25%

0.11%

0.1%

2.09%

0.31%

0.35%

50.19%

8.21%
The heat map suggests that the majority of the methods that are changed in \textit{LinOS} have body-only changes (Body-only column). Out of these methods, most of them remain identical in \textit{AN}. Specifically, 50.19\% of the methods that were changed in \textit{LinOS} have body-only changes in \textit{LinOS} but remain identical in \textit{AN}. This is good news for automatically integrating \textit{LinOS} changes into \textit{AN}, because there are no overlapping changes in \textit{AN}. Combining this information with the information in Table 4.3, it seems that while there is a very high overlap in terms of changed subsystems, there is less change overlap at the method level.

Given that on average almost half the methods changed in \textit{LinOS} (50.19\%) remain identical in \textit{AN}, we further investigate this cell from Table 4.4. We search for the subsystem with the most number of changes in this particular category, across all comparison scenarios. The \texttt{packages/apps/Bluetooth} subsystem in CS7 matches this criteria. Out of 602 \textit{LinOS} changes in this subsystem, 508 of them remained identical in \textit{AO} and had body-only changes in \textit{LinOS}. We randomly sample 20 of these 508 methods and manually inspect them. We realize that for 17 of these methods, there is either a new line added for logging an activity, or an existing line of code regarding a log message was edited. This suggests that the relatively higher number of body-only changes to methods in \textit{LinOS} in CS7 may be because of a new decision towards more thorough logging.

As Table 4.4 shows, the next highest category of changes is the body-only changes in both \textit{LinOS} and \textit{AN}. To better understand this category, we again search for the subsystem with the most changes in this category. It turns out to also be \texttt{packages/apps/Bluetooth}, but this time in CS8 with 170 such changes. Upon further inspection, we realize that similar to the previous case, most of these changes consist of adding a line for logging an activity in \textit{LinOS}, while a different body change has happened in \textit{AO}.

In order to find a more interesting example, we picked the subsystem with the second most number of changes for this category. We find \texttt{SystemUI} for CS6 with 163 such changes. We look for changes that reflect different body-only change overlaps. Figure 4.3 shows code snippets from a sample method
in AO that has gone through body-only changes in both AN and LinOS. The code snippet provides an interesting insight on what kind of changes LINEAGEOS and AOSP make. This method belongs to the BatteryMeter-
View.java class which is responsible for displaying the battery status in the status bar. The method is called when the instance of the class is destroyed. In AO, first the super method is called. Then, a receiver is unregistered. In the next line, the class is removed from the list of callbacks in BatteryController. Finally, an observer is unregistered. In AN, the statements that unregister the receiver and the observer are deleted. However, calls to the super method and BatteryController remain unchanged and two new lines are added. Upon further inspection, we find out that the new lines are related to new features in the new version of AOSP. In LinOS, similar to AN, the call for unregistering the observer was removed. However, unlike AN, the call to BatteryController was also removed and the statement for unregistering the receiver was not removed. A new statement was added in LinOS that removes the class from the list of callbacks in BatteryStateRegistrar class. After exploring the code, we find out that BatteryStateRegistrar is LINEAGEOS’s replacement for AOSP’s BatteryController and adds new options for customizing the battery icon style, something that is not implemented in AOSP.

Answer to S1-RQ2: On average, 50% of changed LinOS methods have body-only changes in LinOS and remain identical in AN. Methods with body-only changes in both LinOS and AN make up an average of 16% of changes in each comparison scenario.

4.3.3 Overall Feasibility of Automation

S1-RQ3 looks at the more general scale, where we are interested to know the percentage of all LinOS changes that have the potential to be automatically integrated into AN. Based on Table 4.1 and our reasoning behind each cell explained in Section 4.1.3, we divide the aggregated analysis results for each comparison scenario into three categories based on the potential for automation: Possible (Green), Might Be Possible (Yellow), Not Possible (Red). Figure 4.4 visualizes the aggregated results of the analysis for each comparison scenario.
: LinOS

```java
public void onDetachedFromWindow() {
    super.onDetachedFromWindow();
    mAttached = false;
    mContext().unregisterReceiver(mTracker);
    mBatteryController.removeStateChangedCallback(this);
    mContext().getContentResolver().
        unregisterContentObserver(mSettingObserver);
    if (mBatteryStateRegistrar != null) {
        mBatteryStateRegistrar.removeStateChangedCallback(this);
    }
}
```

: AO

```java
public void onDetachedFromWindow() {
    super.onDetachedFromWindow();
    mContext().unregisterReceiver(mTracker);
    mBatteryController.removeStateChangedCallback(this);
    mContext().getContentResolver().
        unregisterContentObserver(mSettingObserver);
}
```

: AN

```java
public void onDetachedFromWindow() {
    super.onDetachedFromWindow();
    mContext().unregisterReceiver(mTracker);
    mBatteryController.removeStateChangedCallback(this);
    mContext().getContentResolver().
        unregisterContentObserver(mSettingObserver);
    mDrawable.stopListening();
    TunerService.get(getContext()).removeTunable(this);
}
```

Figure 4.3: Example method in AO from SystemUI subsystem that went through different body-only changes in AN and LinOS.
Figure 4.4: (a) Percentage and (b) count of LinOS changed methods, categorized by feasibility of automated merging into AN for each comparison scenario.
scenario. As the plot suggests, the majority of changes in LinOS can automatically be re-applied onto AN for most of comparison scenarios. On average, out of all LinOS changes across each comparison scenario, 56% (median: 55%) of them belong to the Possible category, 28% (median: 26%) belong to the Not Possible category and the remaining 16% (median: 17%) belong to the Might Be Possible category.

**Answer to S1-RQ3:** On average, 56% of LinOS changes have the potential to be automatically merged into AN. An average of 28% would need developer intervention, while automation might be possible for the remaining 16%.

### 4.4 Discussion

This empirical study is a first step towards investigating the complexity of the Android update problem, and the potential for providing automated support for it. While we were mainly interested in the Android OS, given its size and prevalence, the methodology we used for investigating the problem can be used in other contexts. Our tool chain can be used to investigate the changes that happened in any Java software family where various variants of the software may exist and updates in one direction often need to be made.

In specific to the Android update problem, our empirical investigation showed that while many subsystems are simultaneously modified in LineageOS and AOSP, as many as 63%, on average, of the methods changed in LineageOS are not changed in AOSP. This is good news for solving the Android update problem. If we generally look at all the non-problematic categories in Table 4.1, an average of 56% of method-level changes in LineageOS can potentially be automatically integrated with AOSP changes with every new release. Our analysis of potential for automation was based on numerous discussions by the author of this thesis and another researcher involved in the work, Dr. Sarah Nadi, on whether a tool can handle the merge or integration without the developer’s input. We relied on the semantics of the change to decide on the categorization of the overlap. This is because, as noted in the introduction, there currently does not exist a single off-the-shelf tool that can
correctly and efficiently handle all the types of changes we discussed in this chapter. In order to advance the state of the art and to develop tools and methods that can handle real-world integration and merge problems, such as those illustrated by the Android update problem studied, we dedicate the rest of this section to discuss how we envision using or extending current tooling or techniques to help with the Android update problem, as well as similar problems in other domains or contexts. It is worth mentioning that when discussing types of changes, we consider each of them to be atomic. For example, when discussing methods that were moved, we assume no other type of change were applied to these methods.

### 4.4.1 ‘Possible’ Categories

In this section, we discuss how a practical tool could merge the categories of change in Table 4.1 that have the potential to be automatically merged. The techniques discussed here resemble a three way merge, with AO being the base and AN and LinOS being the other two branches. The idea is that a tool should apply both sets of changes, from AN and LinOS, to AO. We only discuss the table cells with unique combinations of AN and LinOS change types. For example, if we discuss the (AO → AN Method Move) and (AO → LinOS Method Rename) cell, we do not discuss the (AO → AN Method Rename) and (AO → LinOS Method Move) cell.

**AO → AN Identical.** We first consider the methods that were refactored in LinOS (i.e. the method was moved, renamed, inlined, extracted, or its arguments were renamed or reordered). Since the method remained identical in AN, a simple textual merge, e.g., by Git, would be enough to apply the change to that specific method. However, a more thorough way of merging the change is to apply the actual refactoring to make sure that references to the method can be updated accordingly. Since the change detection tools we use, *i.e.*, RefactoringMiner and ChangeDistiller, can already identify the exact type of refactoring that occurred, refactoring parameters can be provided to existing refactoring engines, *e.g.*, in Eclipse [36] or IntelliJ IDEA [54], to automatically apply the change.
Next, we discuss those methods that have an argument change in *LinOS* (i.e. a new argument was added, and old argument was removed, or the type of an argument was changed). Using textual merge tools, the entire method in *AN* could be replaced with its corresponding method in *LinOS*. At this point, the change is merged with no textual conflicts. However, to avoid compilation errors, references to the method also need to be updated. This can be done using static analysis, where all calls to the method could be found and updated to match the new method signature. In the case of argument deletion, updating the references is straightforward. In the case of argument addition, a candidate parameter can be added to the method call by statically finding a variable with the right type in that context. This is obviously a heuristic that may not always work correctly. Another option is to use executable transformations that are extracted from change distilling, similar to the technique suggested by Stevens and De Roover [91]. This way, we can extract an exact transformation script from the change detected in *LinOS*, and execute the exact transformation on *AO*.

Finally, we consider the methods that have changes only in their bodies in *LinOS*. Textual merge tools can simply be used to execute the merge. Since there are no changes in the method’s signature, there is no need to find and update the calls to the method. However, there may be a risk of test failures due to the updated functionality.

*AO → AN Method Move.* The change detection tools we use can provide the refactoring parameters for the method move in *AN*. These parameters can be passed to a refactoring tool to reapply the method move to *AO*. Now, if the same method underwent an argument rename or argument reorder change in *LinOS*, the extracted refactoring parameters from the *LinOS* change can be updated to reflect the new location of the method. A refactoring engine can be then used to reapply the *LinOS* change to the method in its new location in *AO*. If the change in *LinOS* was adding or removing an argument, changing an argument’s type, or a body-only change, textual merge tools can be used to replace the entire method in its new location with the new implementation from *LinOS*. Finally, the tool can utilize static analysis to update all references
to the updated method.

**AO → AN Method Rename.** Similar to the previous category, the tool can extract the refactoring parameters for the rename in **AN**. Using a refactoring engine, this change can be applied to the original method \(m_1\) in **AO**. Now, if part of \(m_1\) was extracted to \(m_2\) in **LinOS**, textual merge can be used to integrate \(m_2\) into **AO**. Next, the modified body of \(m_1\) in **LinOS** can be replaced with its implementation in **AO**. If \(m_1\) had a different change type in **LinOS**, it can be merged to **AO** similar to what we discussed in the previous category.

**AO → AN Method Inline.** Suppose that **AN** inlined the body of \(m_1\) into \(m_2\). In this case, only body-only changes to \(m_1\) in **LinOS** can be automatically integrated. Textual merge can be used to integrate the **LinOS** body-only changes of \(m_1\) into **AO**. The refactoring parameters from the **AN** inline change can then be extracted. These parameters include the signature and position of \(m_2\) that \(m_1\) was inlined into. Using the extracted parameters, a refactoring engine can then be used to inline the updated \(m_1\) into \(m_2\) in **AO**.

**AO → AN Method Extract.** Suppose that part of \(m_1\) was extracted into \(m_2\) in **AN**. Two corresponding refactoring changes in **LinOS** can be automatically integrated: argument rename or argument reorder. A tool could first apply the refactoring change from **LinOS** to \(m_1\) in **AO**. The refactoring parameters for the method extract change from **AN** can then be provided to a refactoring engine to extract the now updated body of \(m_1\) in **AO** into \(m_2\).

**AO → AN Argument Rename.** This change type could be merged if the **LinOS** change type is either argument reorder, argument type change, or body-only change. The tool can first merge the **LinOS** change into **AO** using textual merge tools. If there were any changes to the arguments in **LinOS**, the refactoring parameters related to the argument rename change in **AN** should be updated to reflect the change from **LinOS**. Next, the argument rename change can be applied using a refactoring engine and the new refactoring parameters.

**AO → AN Argument Reorder.** If the method’s change type in **LinOS** is body-only, the new implementation can be replaced in **AO** using textual
merge tools. Then, the method’s signature in AN, that includes the new ordering of arguments, could be replaced with the signature in AO. If the method’s change type in LinOS is argument type change, it can be applied to AO using textual merge tools. Next, the refactoring parameters related to the argument reorder change in AN should be updated to reflect the new argument type LinOS. Using a refactoring engine and the new refactoring parameters, the arguments can be reordered.

4.4.2 ‘Not Possible’ Categories

Unlike the previous category, the merging process of the methods in this category cannot be fully automated. This is mainly because the integration of the two changes in AN and LinOS can only be completed based on the developer’s decision. For example, if a method is deleted in AN and it is moved in LinOS, it is hard to automatically decide if the method should be added back to AN or if the method move should be ignored since the method is deleted. A completely different resolution may also be decided by the developer. For example, she may decide to move the functionality in the refactored LinOS method to a different method in AN. One option would be to at least create some analysis that would calculate potential options and show a preview of these options to the developer. Once the developer decides, the change resolution can potentially be automatically applied.

4.4.3 ‘Might Be Possible’ Categories

For these categories, automation depends on the details of the change. For example, for a method in AO that had one of its arguments renamed in AN and a new argument was added to it in LinOS, automation might not be possible. More specifically, if the name of the new argument in AN is the same as the new name of the renamed argument in LinOS, automation is not possible. However, if this is not the case, a tool may be able to automatically merge these changes. Another case consists of methods in AO that have undergone the same type of change in both AN and LinOS. Integration of such changes can only be automated if the changes are identical. For example, if a method
in AO was moved in both AN and LinOS, we can consider it as a change with possible automation only if it was moved to the same location in both AN and LinOS. A practical solution to changes in this category is to create a list of all change combinations and determine the conditions under which they can be merged automatically.

4.5 Threats to Validity

In this section, we discuss possible threats to the validity of this study.

4.5.1 Construct Validity

Construct validity is concerned with actually measuring the attribute that one intends to measure [58]. In this study, we only cover changes on the method-level. Although there are other types of changes such as changes to classes and their attributes that we are missing, we believe that methods provide a good balance between fine-grained changes and at the same time a meaningful evaluation unit. Evolution studies at the method level have also been performed in several previous work [56].

In our categorization of method changes, we do not consider composite change types. For example, if a method is renamed and also has a change in its body, we only label it as a Method Rename.

Our methodology does not take access level modifiers into account. However, because the static analysis tool that we used, SPOON, can identify access level modifiers, they can be added to our methodology in the future.

In Section 4.2.2, we find mutual repositories between AOSP and LineageOS by looking for exact name matches. Doing so, we would miss repositories in LineageOS that were based on AOSP and later renamed.

4.5.2 Internal Validity

Internal validity is concerned with the effect of additional factors on an observed behavior or attribute [37]. The correctness of the method-level changes we identify relies on the recall and precision of the tools we use: SOURCER-
erCC, RefactoringMiner, and ChangeDistiller. The tools we use are well established and have been previously used in several studies [68], [90]. Additionally, all tools have high reported recall and precision rates, which we provided in Section 3.3.

We additionally manually sampled several of their findings to verify that the tools work as expected. The only issue we faced was with SOURCERERCC, where we found that it misses some Java files during its indexing, and so does not report clones in these files which would lead to inaccuracies in identifying identical changes. However, we could not identify the reason for this. We implemented a workaround for this problem by performing a string comparison between each pair of candidate methods for the body-only change type. If the comparison reveals that the methods are identical, we add them to the identical change type (instead of incorrectly considering them as body-only change).

4.5.3 External Validity

External validity is concerned with whether we can generalize the results outside the scope of our study [37].

We used LineageOS, an open-source community-driven Operating System based on AOSP, in our study. While we cannot generalize our findings to all other phone vendors, our methodology can be applied to other variants of Android, such as proprietary code of phone vendors. Moreover, LineageOS has a large number of active users (50 million in 2015) and a community of more than 1,000 developers. We believe that these qualities render LineageOS as a suitable alternative to a mainstream phone vendor for the purposes of our study. Additionally, the findings of our study already illustrate the problems that arise when independently modified versions of the same software system need to be merged. When searching for subjects, we did consider other variants of Android that are actively maintained, but ruled them out for different reasons. AOKP has a considerable number of users, but is based on LineageOS. MIUI’s source code was not entirely publicly available. While Paranoid Android is fully open-source, we found that it does not have as many releases as LineageOS. Also, the number of changes in each release
were considerably lower than those in LineageOS.

Finally, we do not actually perform any merging or integration, and accordingly, we do not analyze if our anticipated automation may lead to build or test errors. Given the size of Android, any merge attempt using current tools would most likely result in a considerable number of conflicts and resolving them is a daunting task. This is why we decided to first study the nature of changes that occur. Our results show what kinds of change overlap occur in practice, and can be used to guide efforts in improving automation tools. Future work can investigate existing textual merge tools such as Git, existing structural merge tools \cite{7, 8}, as well as the combination of tools mentioned in Section 4.4 to practically apply the integration and analyze the results \cite{36, 54, 91, 94}.

4.6 Summary

In this chapter, we presented the first empirical study of this thesis, which discussed the Android update problem. The problem occurs when a new version of the Android OS is released, and phone vendors need to figure out how to re-apply their proprietary modifications to the new version. We used LineageOS, a community-based variant of Android, as a proxy for a phone vendor. Our results show that when taking the semantics of the changes into account, the majority of overlapping changes between LineageOS and AOSP (56%) have the potential to be automatically merged. We discussed concrete automation opportunities for all categories of change overlaps. Our results are a useful first step for solving the Android update problem, and also for solving the general problem of integrating independently modified variants of a software system. The tooling we used is open-source and can be applied to investigate overlapping changes in any Java system.
Chapter 5

Refactorings and Merge Conflicts

There are several types of conflicts and various reasons why a conflict can occur [71]. Textual conflicts are those that occur when simultaneous changes occur to the same lines in a file, and are the type of conflicts that popular VCSs such as Git detect. For example, one developer may have added a new variable declaration `foo` at line 10 of a given file, while the other developer has added another variable declaration `bar` at the same line. When Git tries to merge both changes, it cannot decide which variable declaration should appear at that line.

As shown in Figure [1.1] refactoring operations have the potential to cause complex textual conflicts. In our empirical study on Android in Chapter [4] we proposed strategies for handling such conflicts. There have been a few other studies that investigated how to deal with refactorings during merging. For example, Dig et al. [33] previously argued that since refactorings cut across module boundaries and affect many parts of the system, they make it harder for VCSs to merge the changed code. They proposed refactoring-aware merging, with the argument that if a merging tool understands the refactorings that took place, it may be able to automatically resolve the conflict and save the developer’s time. In the example in Figure [1.1] their proposed approach would “unapply” the refactoring (i.e., keep `foo` in its old place), apply the new changes to it (i.e., add the new code), and then as a last step, re-apply the refactoring. Other researchers focused on improving code matching and
resolution precision in software merging by considering specific types of refactorings, such as renamings \cite{6}, \cite{61}.

While the above studies propose techniques to deal with refactorings during merging, there have not been any large-scale empirical studies investigating the relationship between refactorings and merge conflicts in the first place. Researchers agree that refactorings may potentially complicate a merge scenario. However, how often does this occur in practice? Do refactorings actually result in more complex conflicts? For example, Dig et al.’s refactoring-aware merging \cite{33} was never evaluated on a large scale. Their technique cannot handle common refactorings such as \texttt{EXTRACT METHOD}. This is because contrary to refactorings such as \texttt{RENAME METHOD}, \texttt{EXTRACT METHOD} refactorings touch method bodies as well as signatures, hence making it difficult to “un-apply” them. If \texttt{EXTRACT METHOD} refactorings are not often involved in merge conflicts, then this is not a main limitation. However, if the opposite is true, then more effort should be invested into improving and extending such refactoring-aware tools.

Understanding the relationship between refactoring and merge conflicts on a large-scale is important to drive researchers’ efforts in the right direction. This study presents the first large-scale empirical study on the relationship between refactorings and merge conflicts. As opposed to related work that looked at a small number of repositories \cite{33} or at a couple of refactoring operations \cite{6}, \cite{61}, our study analyzes close to 3,000 GitHub repositories and uses the state-of-the-art refactoring detection tool, \texttt{REFACTORINGMINER} \cite{94}. In order to understand the relationship between refactoring and merge conflicts, we break down our investigation into the following research questions.

\textbf{S2-RQ1} How often do merge conflicts involve refactored code? Understanding the extent of the impact that refactorings have on merge conflicts would determine the practicality of tools and techniques that assist developers in resolving conflicts that involve refactorings.

\textbf{S2-RQ2} Are conflicts that involve refactoring more difficult to resolve? In order to understand how problematic refactorings are, knowing how
often they are involved in conflicts is not enough per se. A comparison between conflicts that involve refactorings and those that do not will help us better understand differences in complexity.

**S2-RQ3** What types of refactoring are more commonly involved in conflicts? Refactoring-aware merging techniques that rely on “unapplying” refactoring operations would be rendered inefficient if refactoring types that cannot be easily “unapplied”, such as all the extract operations, happen to be involved in conflicts frequently.

### 5.1 Methodology

Our goal is to determine whether refactoring changes are involved in conflicts that occur in Java files. Investigating the relationship between refactorings and conflicts on the commit level or file level may be misleading, since the presence of a refactoring may not be related to the resulting conflict in that commit or file. Therefore, in our work, we investigate the relationship between refactorings and merge conflicts on the *conflicting region* level, because it provides more accurate results than other coarse-grained analysis approaches. The remainder of this section explains this approach in detail.

#### 5.1.1 Overview

Figure 5.1 shows an overview of the steps we follow in our methodology for analyzing a given repository. After identifying merge scenarios with conflicting Java files, we detect all conflicting regions for each scenario. Since we are looking for refactorings that were involved in a conflicting region, we first find all commits after the common ancestor that touched that region, for each merge parent. Next, we use **RefactoringMiner** to detect all refactorings that happened in those commits. Using the location information reported by **RefactoringMiner** for refactoring operations and by **Git** for conflicting regions, we then determine whether a given refactoring was involved in the historical evolution of the conflicting region. All the information we gather in
Figure 5.1: An overview of our methodology for detecting involved refactorings
this process is stored in a MySQL database. Our approach is implemented as a Java tool, which is publicly available [46].

5.1.2 Step 1: Detecting Conflicting Regions

We identify all merge scenarios by finding merge commits in the Git history. Merge commits are commits that have multiple parents. In this work, we focus on merge commits that have only two parents, and record them in our database. We then replay the merge scenario as follows. We first detect the merge parents for each merge commit from the Git history. We then checkout $P_1$, and use the git merge command (with default parameters) to merge $P_2$ into it.

```
  git checkout $P_1
  git merge $P_2
```

By doing so, we can learn (i) whether a given merge scenario is conflicting, as well as (ii) the conflict details, in case it is a conflicting merge scenario. This step is essential because the Git history does not contain such information.

If a merge scenario is conflicting, the git merge command will report the list of conflicting files, as well as their conflict type (See Section 3.1.2). Using this list, we record the conflicting Java files and their conflict type to the database, if any. For Java files with content conflict type, we detect all conflicting regions by using the git diff command. When in a conflicting state, this command will report all conflicting regions, along with the corresponding location of each region in both merge parents. Because this command reports a few lines before and after the conflicting region, we use the -U0 parameter to remove these extra lines. We record this information in the database.

```
  git diff -U0
```

Step 1 in Figure 5.1 shows an example of the conflicting region produced by running the above git diff command. The three pairs of numbers between the @@@ symbols denote the conflicting region. The first pair of numbers corresponds to the region in $P_1$, while the second pair corresponds to $P_2$. The third pair of numbers is the conflicting region in the conflicting merged file with the markers. In each pair, the first number is the line number where the region
begins and the number after comma is the length of that region. Because we
are interested in the location of the conflicting region in each merge parent,
we only record the first two pairs of numbers for each conflicting region.

5.1.3 Step 2: Detecting Evolutionary Changes

In the next step of our methodology, we track the historical evolution of a
given conflicting region between the common ancestor and each merge parent.
Using this information, we can determine if a refactoring was involved in any
of these evolutionary changes which later led to a conflict. We use the `git log`
command to perform this task. `git log` is a useful and versatile command
thanks to the different parameters it accepts. Using the `-L` parameter along
with a revision range, it will report all commits in the revision range that have
touched the given file in the specified location. For a given conflicting region,
we run `git log` once for each merge parent, with the `-L` parameter set to the
corresponding location of the conflicting region in that parent (extracted in
Step 1), and the revision range set between that merge parent and the common
ancestor. For example, the revision range `P2..P1` includes all commits that
are reachable from `P1` and not reachable from `P2`, which is equivalent to the
commits between `P1` and the common ancestor of `P1` and `P2`.

```
git log -L start_{P1},end_{P1}:file P2..P1
git log -L start_{P2},end_{P2}:file P1..P2
```

This command outputs all commits that have touched the specified location
as well as the corresponding location information for each commit. In our ex-
ample in Figure 5.1 (Step 2), running the above commands returns the black-
dotted commits. We call these commits *evolutionary commits* since they are
involved in the evolution of the conflicting region. The rectangles connected
to these commits contain the reported location information. The two number
pairs between the @@ symbols correspond to the location of the conflicting
region before and after that commit, respectively. For the top commit, for
example, the conflicting region can be found at line number 60 before this
commit, and at line 65 after this commit. We save this information in the

[^1]: https://git-scm.com/docs/git-log
5.1.4 Step 3: Detecting Refactorings

In this step, we use RefactoringMiner to detect the refactoring operations taking place in the commits that were involved in the evolution of conflicting regions (i.e., in the evolutionary commits identified in Step 2). In addition to the refactoring type, RefactoringMiner reports the files and the exact code ranges (with line numbers) that were touched by a refactoring operation. We store at least two code ranges for a refactoring change: one code range corresponds to the refactored code element before refactoring, and the other corresponds to the element after refactoring. Table 5.1 provides a summary of the code ranges we store in the database for different refactoring types.

5.1.5 Step 4: Detecting Involved Refactorings

In the final step of our methodology, we identify the refactorings that have affected the evolution of conflicting regions. In other words, we are trying to determine if a refactoring touched an evolutionary change that later lead to a conflict. Using the code range information that we have for both refactorings (Step 3) and evolutionary changes to conflicting regions (Step 2), we determine if there is an overlap between them. We consider a refactoring and evolutionary change as overlapping if they have at least one line in common, either in their old-commit code ranges or in their new-commit code ranges. For example, line numbers 34:40 and 38:44 are overlapping while line numbers 34:40 and 42:57 have no overlaps. We call such refactorings that have overlapping code ranges with an evolutionary change involved refactorings, since they are involved in the changes that are related to the conflicting region. In the example of Figure 5.1, Step 4 shows that the refactoring in commit #1 would not be considered as an involved refactoring, while the refactoring in commit #2 would be considered so.

https://github.com/tsantalis/RefactoringMiner, used as of commit 46c80ad
<table>
<thead>
<tr>
<th>Code Element</th>
<th>Refactoring Type</th>
<th>Stored Code Range Corresponding to Old Commit</th>
<th>New Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Field</td>
<td>Pluck Up Field, Push Down Field, Move Field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Method</td>
<td>Inline Method</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>Extract Superclass/Interface</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pull Up Field, Push Down Field, Move Field</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Stored code ranges for each refactoring type
5.2 Evaluation Setup

In this section, we explain the setup we use for exploring the relation between refactorings and conflicts, based on the methodology from Section 5.1.

5.2.1 Repository Selection

The first step of our evaluation setup is to determine the set of Git repositories we will run our analysis on. GitHub is a source-code hosting service that contains over 85 million software repositories. Many software engineering researchers use GitHub to obtain a set of repositories and analyze them for their studies [27], [47], [51], [57], [86], [95], [103]. However, considering the public nature of GitHub, including random repositories without employing a filtering process could lead to misleading findings. For example, many students use GitHub to upload the source code of their course works and programming assignments. Ideally, we want the conclusions of our study to be indicative of how refactorings affect merge conflicts in realistic development setups.

Munaiah et al. [74] studied GitHub repositories and proposed two classifiers (Score-based and Random Forest) that determine whether a given repository is a well-engineered software project. Based on their results, the Random Forest classifier has a higher precision rate. Using their dataset of 1,857,423 repositories, we filter out projects that are not labeled as well-engineered by the Random Forest classifier. Additionally, given the focus of our work, we only consider repositories that are implemented in Java. In our final filtering step, to further ensure the quality of the repositories we pick, we only include repositories with 100 or more stars on GitHub. This leaves us with a dataset of 2,955 repositories, which we use for our study. However, we find that 30 repositories from this list are no longer accessible with the provided GitHub URL, and so we exclude them from the analysis. Thus, we run our methodology on the final list of 2,925 repositories.

https://github.com/features
5.2.2 RefactoringMiner Settings

When using RefactoringMiner, we find that some commits may take longer to process, sometimes leaving the process to hang. Given the scale of our study, we need to ensure that the analysis terminates in a timely manner. Accordingly, we enforce a timeout of 4 minutes on RefactoringMiner. If RefactoringMiner does not terminate within 4 minutes on a given commit, we terminate the process and skip this commit.

5.2.3 Running Environment

For running the analysis, we used 12 threads on a machine with 16 CPU cores at 3.4GHz, 128 gigabytes of memory, a solid-state storage device, and a 1 Gbps Internet connection. Each thread runs the entire process for a repository. Analyzing all repositories took a total of 27 hours.

5.3 Results

In this section, we report the results of running our methodology from Section 5.1 on the 2,925 repositories described in Section 5.2. When running the analysis, we find that one repository (platform_frameworks_base) took a much longer time to process, due to its unusually high number of merge scenarios: this repository has 281,251 merge scenarios, while all other projects in our dataset have 729,060 combined. We decided to skip this repository in our analysis because this uncommon irregularity might skew our results. Thus, all the results provided in this section are based on the analysis of the remaining 2,924 repositories. We first report some descriptive statistics of the data collected, and then proceed to answer each of the research questions we presented in the introduction.

5.3.1 Descriptive Statistics of Collected Data

Table 5.2 provides a summary of the collected data after running our methodology on our set of 2,924 repositories, including the mean and standard de-
Table 5.2: Statistics for merge scenarios, merge conflicts, and refactorings

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Total</th>
<th># of Corresponding Repositories</th>
<th>Per Repository Mean</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Scenario</td>
<td>729,060</td>
<td>2,606</td>
<td>279.76</td>
<td>1,690.83</td>
</tr>
<tr>
<td>Conflicting Merge Scenario (CMS)</td>
<td>63,826</td>
<td>1,753</td>
<td>36.40</td>
<td>144.36</td>
</tr>
<tr>
<td>CMS with Java Conflicts</td>
<td>36,988</td>
<td>1,424</td>
<td>25.97</td>
<td>91.82</td>
</tr>
<tr>
<td>Conflicting Region</td>
<td>258,956</td>
<td>1,403</td>
<td>184.57</td>
<td>767.38</td>
</tr>
<tr>
<td>Evolutionary Commit</td>
<td>657,726</td>
<td>1,396</td>
<td>471.15</td>
<td>2,073.24</td>
</tr>
<tr>
<td>Refactoring in Evolutionary Commits</td>
<td>248,652</td>
<td>1,136</td>
<td>218.88</td>
<td>783.61</td>
</tr>
</tbody>
</table>

We find that 2,606 of the repositories we analyzed contained merge scenarios (a total of 729,060 merge scenarios), out of which 1,753 repositories had at least one conflicting merge scenario (a total of 63,826 conflicting merge scenarios). Out of those, 1,424 repositories had at least one conflicting merge scenario (CMS) that included a conflicting Java file (a total of 36,988 CMSs with conflicting Java files). Since not all conflict types can have conflicting regions, a fewer number of repositories, 1,403, have conflicting regions (a total of 258,956 conflicting regions). Furthermore, 1,396 of those repositories have historical evolutionary changes for their conflicting regions, with these changes occurring in a total of 657,726 commits. We explain the discrepancy between the number of repositories with conflicting regions and repositories with evolutionary commits in Section 5.4.1. The results we present in the rest of this section are thus based on the 1,396 repositories for which we were able to extract historical evolutionary changes for their conflicting regions. Note that the last row in Table 5.2 shows the number of refactoring operations detected by RefactoringMiner in evolutionary commits.

As an additional data point not shown in the table, the 36,988 conflicting merge scenarios with Java conflicts we collected contain 157,422 conflicting Java files. The conflicts in these Java files can belong to any of the conflict types discussed in Section 3.1.2. Out of these, 99,846 files (i.e., 63%) are
content conflicts. This suggests that content conflicts, which is the focus of our work, represent the majority of conflicts that developers face in practice.

5.3.2 How often do merge conflicts involve refactored code?

Data used for RQ

We answer this RQ by checking whether a code change that led to a conflict involved a refactoring change. As explained in Section 5.1, we use the term involved refactoring to describe a refactoring that happened in an evolutionary change and which overlaps with the conflicting region. A conflicting merge scenario can have multiple conflicting regions. If at least one of the conflicting regions in a conflicting merge scenario contains involved refactorings, we consider that merge scenario as one that contains involved refactorings.

Findings

We find that there are 8,155 conflicting merge scenarios that contain involved refactorings. We know from Table 5.2 that there is a total of 36,988 merge scenarios with conflicting Java files, which means that 22% of these merge scenarios involve refactorings. On the conflicting region level, we find that 28,670 (i.e., 11%) of the 258,956 conflicting regions from Table 5.2 have involved refactorings.

Answer to S2-RQ1: 22% of merge scenarios with at least one conflicting Java file involve refactorings. More precisely, 11% of conflicting regions have at least one involved refactoring.

Implications

Since there are no previous studies that investigated the extent of refactorings in merge conflicts, or other types of semantic code changes in merge conflicts, we have no point of reference to compare our findings to. However, previous work by McKee et al. [70] showed that when resolving merge conflicts, practitioners’ top needs include: (a) understanding the code involved in the merge conflict, and (b) tools to help them explore the project history during the pro-
cess of resolving conflicts. While 22% might not seem like a high number, and we cannot conclude that refactorings are involved in the majority of merge conflicts, our findings provide good news for addressing practitioners' requests from McKee et al.'s study. Since refactoring detection in commit history has now become precise and scalable, this means that, based on our results, researchers can provide developers with tools to explore the history and interpret changes in a little less than a quarter of conflicting merge scenarios, thus helping them to resolve conflicts faster. On the level of a given conflicting scenario, such tool support can be provided for approximately 11% of the conflicting regions. It should be emphasized that in our study, we considered only a subset of all possible refactoring types, because RefactoringMiner supports only 15 out of the 72 refactoring types described in Fowler's catalog [43]. We conjecture that the aforementioned percentages would be potentially even larger if more refactoring types were considered.

5.3.3 Are conflicts that involve refactoring more difficult to resolve?

Data used in RQ

Previous work on software merging used the number of conflicting lines as a measure of the complexity of a conflict [7], [8]. Recent work also confirms that the number of conflicting lines is one of the top factors that affects the developers' perception of the difficulty of a conflict [70]. Based on the above previous work, we use the number of conflicting lines, in other words the size of conflicting region, as a proxy for describing the difficulty of a merge conflict. As an additional measure of difficulty, we also look at how refactorings can affect the number of evolutionary commits for conflicting merge scenarios. Since program comprehension is a traditionally complex and time consuming task [25], we assume that the more evolutionary changes a merge scenario has, the more complex resolving a merge conflict will be, since more changes need to be understood.
Figure 5.2: Distribution of conflicting region size with and without involved refactorings
Findings

Figure 5.2 shows the size of all conflicting regions in our study. The left box plot contains conflicting regions with involved refactorings, while the right box plot contains the remaining conflicting regions. The orange lines mark the median and the green triangles show the mean. The figure shows that there are a few conflicting regions without refactorings that are larger than any conflicting region with refactoring (those with >10,000 lines). However, this does not hold on average. The mean and median for conflicting regions with involved refactorings are 39.63 and 12 respectively. The same values are lower for conflicting regions without involved refactorings, with a mean and median of 26.63 and 7, respectively. The unpaired Wilcoxon rank-sum test shows that these distributions are statistically different ($p$-value = 0.00). Additionally, a 95% confidence interval for the difference between the two population medians is between 3 and infinity, suggesting that the median of the size of conflicting regions with involved refactorings is at least 3 lines larger than the median size of the remaining conflicting regions. For measuring the effect size, we use $r = \frac{Z}{\sqrt{N}}$ where $Z$ is the test statistic and $N$ is the number of samples [39]. The effect size is 0.14 which can be interpreted as small.

Figure 5.3 shows the distribution of the number of evolutionary commits for each conflicting merge scenario. The left box plot contains conflicting merge scenarios with at least one involved refactoring change in their evolutionary commits. The right box plot contains the remaining conflicting merge scenarios. Similar to Figure 5.2, the orange line shows the median and the green triangle marks the mean. As Figure 5.3 suggests, conflicting merge scenarios with involved refactorings have a larger number of evolutionary commits (median: 5%, mean: 10.43%) compared to conflicting merge scenarios with no involved refactoring changes (median: 2%, mean: 3.46%). Furthermore, the Wilcoxon rank-sum test shows that this difference is significant ($p$-value = 0.00). Using the $r = \frac{Z}{\sqrt{N}}$ formula, we find that the effect size is 0.34 which can be interpreted as medium.
Figure 5.3: Number of evolutionary commits per merge scenario with and without involved refactorings
Answer to S2-RQ2: Conflicting regions that involve refactorings tend to be larger (i.e., more complex) than those without refactorings. Furthermore, conflicting merge scenarios with involved refactorings include more evolutionary changes (i.e., changes leading to conflict) than conflicting merge scenarios without involved refactorings.

Implications

Our findings show that conflicting regions that involve refactoring operations are indeed more complex than conflicting regions without involved refactorings. While 3 extra conflicting lines may not seem like a big difference, recall that the median size of a conflicting region is already small (7 lines), so 3 lines represents an almost 50% increase. Additionally, our results suggest that resolving merge conflicts with involved refactorings may be more difficult, since they typically involve more evolutionary changes for the developer doing the resolution to understand. Our findings provide great motivation for refactoring-aware merging tools and techniques that can help developers in the merging process.

5.3.4 What types of refactoring are more commonly involved in conflicts?

Data Used for RQ

We consider 15 types of refactorings in our work. Not all of them necessarily occur with the same rate, and each refactoring type might impact conflicting regions differently. Understanding how often each refactoring type affects merge conflicts is important for any future tool support, especially for refactoring-aware merging tools and techniques.

When looking for differences in the distribution of each refactoring type among all involved refactorings, it is important to take into account the “typical” distribution of refactorings types as well, i.e., how often each refactoring type occurs in general. This way, we can observe if there are any discrepancies between the distribution of a given refactoring type among involved refactorings vs. in general. Specifically, it would be interesting to find the refactoring types that appear more often as involved refactorings when compared to their
Figure 5.4: Percentage of each involved refactoring's type per project.
general distribution. Such cases indicate particularly problematic refactorings that are involved in merge conflicts. However, in our methodology, we do not collect information about all refactorings that happen in each repository. As described in Section 5.1.4, we detect refactoring operations only in evolutionary commits. Since not all detected refactoring operations are involved refactorings, we use the distribution of all detected refactorings in all evolutionary commits as a proxy for the general distribution of refactorings in our data.

Findings

Figure 5.4 shows how common each refactoring type is across all projects. It provides two different distributions for each refactoring type: involved refactorings and overall refactorings. Every project has two data points in each violin plot, representing a refactoring type. The y-axis is the percentage of refactorings corresponding to the refactoring type. The width of each plot at a given percentage shows the number of projects with that percentage. For example, suppose a project has a total of 5 refactorings in its evolutionary commits, two Rename Method refactorings and three Move Class refactorings. Also, assume that the two Rename Method refactorings are involved in conflicting regions. This project will be represented by 30 data points in the figure, two points for each refactoring type. For the violin plots corresponding to involved refactorings, all of the points for this project will have a value of zero, except the point corresponding to Rename Method which will be 100%. For the violin plots corresponding to overall refactorings, the points for Rename Method and Move Class will have a value of 40% and 60%, respectively, and the points for the remaining refactoring types will be zero.

We use a two-sided paired Wilcoxon signed-rank test to compare the distributions of overall refactorings and involved refactorings, for each refactoring type. We use a Benjamini & Hochberg (BH) p-value adjustment measure to account for multiple comparisons, and use $\alpha = 0.05$. We use the $r$ value for effect size, which is recommended by Fields [39] for the Wilcoxon signed-rank
Table 5.3: Wilcoxon signed-rank paired test results between overall and involved refactorings. When involved refactorings are more than overall refactorings, the direction of difference is ↑, and ↓ otherwise. Statistically significant results (p < 0.05) are shown in bold, with highlighted rows being specifically of interest.
test. We interpret it as small ($\geq 0.1$), medium ($\geq 0.3$), and large ($\geq 0.5$) \[39\]. To find the direction of the difference, we compare the means and interquartile ranges of the distributions. We show the results in Table 5.3. The third column shows that 12 refactoring types have $p$-values lower than 0.05 (highlighted in bold), meaning that involved and overall refactorings in these types have a different distribution.

Out of these 12 types, involved refactorings have higher percentages for Extract Interface, Extract Method, and Extract Superclass. However, the effect size is negligible for Extract Superclass and Extract Interface and is small for Extract Method.

Answer to S2-RQ3: Extract Method is more involved in conflicts than its typical overall frequency, with a small effect size. Extract Interface and Extract Superclass are also more involved in conflicts, but with negligible effect sizes.

Implications

Our results suggest bad news for existing refactoring-aware merging tools. As mentioned in the introduction of this chapter, Dig et al.’s undo/redo technique \[33\] cannot handle extract refactorings. This calls for more sophisticated refactoring-aware merging tools that can handle such cases.

With the exception of Move Class, the effect size of the remaining refactorings in the other direction is small. The larger effect size for Move Class might have to do with the fact that Git detects file move and rename operations. Since a Java public class is represented in a single file, the Move Class refactoring is essentially a file move operation for public Java classes. Now, if a class is moved in one branch, while its content was edited in another branch, Git can automatically merge the content change since it is aware of the move. Hence, a fewer number of Move Class refactorings end up being involved in conflicts, based on our region-based methodology.

5.4 Threats to Validity

In this section, we discuss possible threats to the validity of this study.
5.4.1 Construct Validity

Construct validity is concerned with actually measuring the attribute that one intends to measure [58].

In Table 5.2, the number of repositories with evolutionary commits is less than the number of repositories with conflicting regions. Upon further investigation, we found that about 4% (11,312 out of 258,956) of conflicting regions do not have corresponding evolutionary commits. After manual sampling, we found that this occurs for merge scenarios that contain nested merge scenarios within their evolutionary changes. Since we do not go back and examine the previous histories of these nested merge commits, it means that for some conflicting regions, we miss the changes that may have caused the conflict, if those changes were caused by a merge commit. However, since we analyze every merge scenario in a repository, the inner merge scenarios will be individually analyzed and we will collect their evolutionary commits and corresponding refactorings. Any missed involved refactorings, because of these nested merge cases simply means that our reported stats are a lower bound of the actual involvement of refactorings in merge conflicts.

When looking for merge commits, we only consider merge commits with two parents, while in reality a merge commit can have more than two parents. However, merge commits with more than 2 parents happen rarely in practice.

In our methodology, we looked for refactoring operations that were involved in evolutionary commits that led to conflicts. However, it is not easy to determine whether an involved refactoring indeed caused a conflict, or even more so, whether it was the sole cause of that conflict. This is because refactorings are usually interspersed with other type of changes [75]. As a result, determining whether a conflict was caused by a refactoring or other changes that were tangled with the refactoring is a difficult task. This is why we conservatively say that these refactorings were involved in conflicts and refrain from claiming that they directly caused the conflicts.
5.4.2 Internal Validity

Internal validity is concerned with the effect of additional factors on an observed behavior or attribute [37].

Any inaccuracies in our tooling may lead to wrong results. To mitigate that, we use the state-of-the-art refactoring tool, RefactoringMiner, which has high precision. Additionally, we manually reviewed samples of our results throughout our experiments. In terms of our methodology for calculating involved refactorings, we manually validated samples of our results and also publish our tooling and findings in our online artifact page [46] to facilitate reproducability.

As mentioned in Section 5.2, when using RefactoringMiner to detect refactorings in a given commit, we employ a 4-minute timeout. We keep a record of every time RefactoringMiner takes more than 4 minutes and the process is terminated. Out of 115,911 commits that we analyzed with RefactoringMiner, only 949 of them (0.81%) reached this timeout. This is a very small percentage and does not pose a serious threat to the validity of our results.

5.4.3 External Validity

External validity is concerned with whether we can generalize the results outside the scope of our study [37].

Our study focuses only on Java projects. By limiting our subject systems to well-engineered software projects, we made our results more indicative of realistic development setups. However, since we were limited to open-source software systems and we did not have access to closed-source enterprise Java projects, we cannot claim that our findings can be generalized to all Java software systems. Nonetheless, the large number of subject systems we use (almost 3,000) suggest that our findings are common in open-source projects.

Finally, RefactoringMiner is able to detect only 15 refactoring types out of the 72 refactoring types described in Fowler’s catalog [43], so we cannot generalize beyond the refactoring types we consider. However, the investigated
refactoring types are among the most frequently applied types [76]. We conjecture that a study considering a larger set of refactoring types would possibly find an even stronger involvement of refactoring operations in merge conflicts.

5.5 Summary

In this chapter, we presented our second empirical study on refactoring changes and merge conflicts. This study, to the best of our knowledge, is the first large-scale empirical study to understand the relationship between refactorings and merge conflicts. We studied almost 3,000 well-engineered open-source Java repositories. Our results show that refactoring operations are involved in 22% of merge conflicts. Moreover, we find that conflicts that involve refactorings are often more complex. Our findings suggest that merging tool support that understands refactoring can have a positive impact in practice, while existing tools that rely on reverting refactoring operations during merging may need improvement.
Chapter 6

Conclusion

In this thesis, we performed two empirical studies to understand the challenges that developers face in software merging and identify possible directions to help them overcome these problems.

We first investigated the integration challenges in forked repositories. We focused on the Android update problem. The problem occurs when a new version of the Android OS is released, and phone vendors need to figure out how to re-apply their proprietary modifications to the new version. This leads to late adoption of the new version of Android by phone vendors, which may lead to problems associated with using an outdated version of the OS, such as security vulnerabilities.

By studying eight versions of a community-based variant of Android called LineageOS, which we use as a proxy for a phone vendor, we analyzed the details and overlap of the changes applied in LineageOS versus those in Android. Based on the semantics of the changes, we categorized whether the overlapping changes have the potential to be automatically integrated or not.

Our results show that both LineageOS and Android often change similar parts of the OS. We find that the subsystems related to the settings in Android as well as the user interface are mutually changed by LineageOS and Android. On the other hand, when taking the semantics of the changes into account, we find that the majority of overlapping changes (56%) have the potential to be automatically merged. We discussed concrete automation opportunities for all categories of change overlaps.
This study is a useful first step for solving the Android update problem, and also for solving the general problem of integrating independently modified variants of a software system. Researchers can use our tool chain and employ our methodology to other variants of Android, or other similar independently modified projects. This enables them to analyze the frequency and type of changes that are applied to an independently modified repository vs. the main repository it was forked from.

Future research can investigate the integration solutions we discussed in practice. This can lead to the implementation of practical tools that can assist developers when they face integration challenges.

Our results suggest that apart from changes that are currently automatically integrated by textual merge tools, refactoring changes comprise the majority of changes that have the potential to be automatically integrated. This lead us to study the relationship between refactorings and merge conflicts in more details.

We performed, to the best of our knowledge, the first large-scale empirical study to understand the relationship between refactorings and merge conflicts. We studied almost 3,000 well-engineered open-source Java repositories. Using RefactoringMiner, we detected refactoring operations that were involved in merge conflicts.

Our results show that refactoring operations are involved in 22% of merge conflicts. We also find that conflicts that involve refactorings are often more complex. Furthermore, we find that existing tools that rely on reverting refactoring operations during merging may need improvement, since the Extract Method refactoring is involved in more conflicts than its typical frequency.

This study provides empirical motivation for future research in refactoring-aware software merging and shows that merging tool support that understands refactoring can have a positive impact in practice. Such tool support can vary from helping the developer understand the changes that led to the conflict to automatically resolving the conflict for them.
References


