MONASH University

Compatibility Issues in Android: Characterization and Detection

Pei Liu

Doctor of Philosophy

A Thesis Submitted for the Degree of Doctor of Philosophy at
Monash University in 2023
Faculty of Information Technology

Copyright notice

I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner's permission.

Abstract

With over 500,000 commits and more than 700 contributors, the Android platform is undoubtedly one of the largest industrial-scale software projects and has become one of the most popular and successful open-source operating systems in the last decades. Benefiting from its open-source attribute, device vendors and OS providers offer customized versions of the Android OS to provide special treatment for customers. To meet customers' various needs, such as online shopping, video streaming, taking classes, etc., developers build their Android apps via the Android APIs released alongside Android OS. Unfortunately, because the Android platform and its customizations evolve at an extremely rapid pace, app developers need to continually monitor API changes to avoid compatibility issues (i.e., issues that prevent apps from working as expected when running on different devices running different API versions) in their apps. Despite a large number of studies on compatibility issues in the Android ecosystem, the research community has not yet investigated the compatibility issues that might be induced by the APIs with the same signatures but different implementations in other official Android versions or third-party customizations.

This Ph.D. work focuses on the characterization and detection of compatibility issues. To be more specific, we present three different studies to explore the introduction of the problematic APIs and their possible severe ramifications on Android apps, summarize the advantages and shortcomings of the state-of-the-art detection tools, and propose a tool, *AndroMevol*, to systematically harvest incompatible APIs to enhance the capability of existing detection tools and further guide for future research in the area of compatibility issue. In summary, the main contributions of the dissertation are:

- **ANDROSEA**. App developers heavily rely on the provided APIs to develop their Android apps. To avoid compatibility issues (i.e., issues that prevent apps from working as expected when running on different versions of the API) in their Apps, they always take extra efforts to continuously monitor API changes on the official development site. Despite a large number of studies on compatibility issues in the Android ecosystem, the research community has not yet investigated issues related to silently-evolved methods (SEMs). These methods are functions whose behavior might have changed but the corresponding documentation did not update accordingly, which would mislead developers developing Android apps with unexpected failures emerging when running the apps. To shed light on this type of issue, we proposed an automatic approach embodied in the tool ANDROSEA to identify and characterize such SEMs across ten consecutive versions of Android releases.

- **Empirical Study**. To satisfy varying customer needs, device vendors and OS providers often rely on the open-source nature of the Android OS and offer customized versions of the Android OS. When a new version of the Android OS is released, device vendors and OS providers need to merge the changes from the Android OS into their customizations to account for its bug fixes, security patches, and new features. As developers of customized OSs may have modified code locations that were also modified by the Android OS developers, the merging task can be characterized by conflicts, which can be time-consuming and error-prone to resolve. To resolve method-related conflicts, customized OS maintainers may introduce inconsistencies compared to the official methods, which can eventually lead to compatibility issues in Android apps. To provide more insight into this critical aspect of the Android ecosystem, we analyze how often the developers from the customized OSs merge changes from the Android OS, how often they experience textual merge conflicts, and the characteristics of these conflicts and their impact on Android apps.

- **Replicability Study**. Compatibility issues in Android apps are mainly caused by the fast evolution of the system and the various customizations maintained by different smartphone manufacturers and third-party OS providers. Despite many efforts to mitigate their impact via approaches to automatically pinpoint compatibility issues, it is still unclear whether this objective has been achieved, and whether the existing approaches can reliably be leveraged to identify compatibility issues in the wild. We, therefore, first conduct a literature review on this topic to identify all the available approaches and summarize their typical steps in issue detection. In this process, we find that two separate steps, including gathering incompatible APIs and determining compatibility issues (induced by such incompatible API's invocation), are commonly leveraged in the nine identified detection approaches We attempt to reproduce and compare them based on their original and newly introduced datasets containing real-world apps with compatibility issues. To improve the performance of the existing detection tools and guide future research in the area of issue detection, we propose *AndroMevol* to systematically gather incompatible APIs from six OS brands.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: _____

Name: Pei Liu

Date: 03/04/2023

Thesis including published works declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes 2 original papers published in peer-reviewed journals and conferences and 1 submitted publication. The core theme of the thesis is requirements changes in software development. The ideas, development, and writing up of all the papers in the thesis were the principal responsibility of myself, the student, working within the Department of Software Systems and Cybersecurity under the supervision of Professor John Grundy and Dr. Li Li.

The inclusion of co-authors reflects the fact that the work came from active collaboration between researchers and acknowledges input into team-based research.

In the case of Chapter *3, 4, 5* and *6* my contribution to the work involved the following: *(See next page)*

TABLE 1: Publication Author Contributions

Thesis Chapter	Publication Title	Status *(published, in press, accepted or returned for revision, submitted)*	Nature and % of Student Contribution	Co-author name(s), Nature and % of Co-authors's Contribution*	Co-Author(s), Monash student Y/N*
3	Identifying and Characterizing Silently-Evolved Methods in the Android API	Published	65%. Concept, conducting the experiments, and writing the manuscript	1. Li Li, Concept and input into the manuscript, 10%. 2. Yichun Yan, Experiments, 5%. 3. Mattia Fazzini, Concept and input into the manuscript, 10%. 4. John Grundy, Supervised the study and input into the manuscript, 10%.	1. No 2. No 3. No 4. No
4	Do Customized Android Frameworks Keep Pace with Android?	Published	70%. Concept, conducting the experiments, and writing the manuscript	1. Mattia Fazzini, Concept and input into the manuscript, 10%. 2. John Grundy, Supervised the study and input into the manuscript, 10%. 3. Li Li, Concept and input into the manuscript, 10%.	1. No 2. No 3. No
5	Automatically Detecting Incompatible Android APIs	Accepted	40%. Concept, conducting the experiments, and writing the manuscript	1. Yanjie Zhao, Concept, experiments, and input into the manuscript, 30% 2. Haipeng Cai, Concept, and input into the manuscript, 5%. 3. Mattia Fazzini, Concept and input into the manuscript, 10%. 4. John Grundy, Supervised the study and input into the manuscript, 5%. 5. Li Li, Concept and input into the manuscript, 10%.	1. Yes 2. No 3. No 4. No 5. No

I have not renumbered sections of submitted or published papers in order to generate a consistent presentation within the thesis.

Student name: Pei Liu

Student signature:

Date: 27/02/2023

I hereby certify that the above declaration correctly reflects the nature and extent of the student's and co-authors' contributions to this work. In instances where I am not the responsible author I have consulted with the responsible author to agree on the respective contributions of the authors.

Main Supervisor name: John Grundy

Main Supervisor signature:

Date: 20/04/2023

Publications during enrolment

This thesis contains ten publications that are either published (9) or accepted (1).

Publications included in this thesis:

1. **Pei, Liu**, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. "Identifying and characterizing silently-evolved methods in the android API." In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 308-317. IEEE, 2021.
2. **Pei, Liu**, Mattia Fazzini, John Grundy, and Li Li. "Do customized Android frameworks keep pace with Android?." In Proceedings of the 19th International Conference on Mining Software Repositories, pp. 376-387. 2022.

Accepted manuscripts included in this thesis:

1. **Pei, Liu**, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. "Automatically Detecting Incompatible Android APIs." ACM Transactions on Software Engineering and Methodology (Accepted).

Other publications during enrolment (not included in this thesis):

1. **Pei, Liu**, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. "Automatically detecting api-induced compatibility issues in android apps: a comparative analysis (replicability study)." In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 617-628. 2022.
2. **Pei, Liu**, Qingxin Xia, Kui Liu, Juncai Guo, Xin Wang, Jin Liu, John Grundy, and Li Li. "Towards automated Android app internationalisation: An exploratory study." Journal of Systems and Software 197 (2023): 111559.
3. Xiaoyu, Sun, Xiao Chen, Yanjie Zhao, **Pei, Liu**, John Grundy, and Li Li. "Mining android api usage to generate unit test cases for pinpointing compatibility issues." In 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-13. 2022.
4. **Pei, Liu**, Xiaoyu Sun, Yanjie Zhao, Yonghui Liu, John Grundy, and Li Li. "A First Look at CI/CD Adoptions in Open-Source Android Apps." In 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-6. 2022.

5. Yanjie, Zhao, Li Li, Xiaoyu Sun, **Pei, Liu**, and John Grundy. ”Code implementation recommendation for Android GUI components.” In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 31-35. 2022.

6. Yanjie, Zhao, Li Li, Xiaoyu Sun, **Pei, Liu**, and John Grundy. ”Icon2Code: Recommending code implementations for Android GUI components.” Information and Software Technology 138 (2021): 106619.

7. **Pei, Liu**, Li Li, Yanjie Zhao, Xiaoyu Sun, and John Grundy. ”Androzooopen: Collecting large-scale open source android apps for the research community.” In Proceedings of the 17th International Conference on Mining Software Repositories, pp. 548-552. 2020.

Acknowledgements

Time flies. I still remember the day when I first came to Melbourne and that day feels like yesterday. The road to earning a Ph.D. degree is a long journey that I cannot accomplish by myself alone. With the greatest fortune, I have been helped, encouraged, and supported by many brilliant people. I would like to thank all these people and the amazing scientific journey.

First, I would like to express my sincere gratitude to my supervisors, Prof. John Grundy and Dr. Li Li for their enthusiastic inspiration and thoughtful guidance throughout my Ph.D. study. They have provided me with a great deal of assistance in my life and study. The enthusiasm and passion Dr. Li expressed during our weekly meeting for research have really impressed and influenced me. He taught me how to identify and solve real interesting and worthwhile research problems in the realm of software engineering and how to present research results clearly in scientific research reports and conference papers. Prof. John Grundy helped me a lot in research idea formulation and attractive presentation preparation. His optimism helped me face and overcome my weaknesses. Without their patience and commitment, I would never ever have had a chance to experience the fascinating Ph.D. journey.

Secondly, I would like to thank my panel members, including A/Prof. Yuan-Fang Li, A/Prof. Aldeida Aleti, A/Prof. Ron Steinfeld, and Dr. Jiangshan Yu all from Monash University, and co-authors, including A/Prof. Haipeng Cai from Washington State University, Pullman, and Asst. Prof. Mattia Fazzini from the University of Minnesota, Minneapolis. It is my great honor to have these brilliant people as my milestone panel members and research paper co-authors. Their valuable suggestions have significantly improved my research work making my work more solid and impactful.

Furthermore, I would extend my gratitude to my friends and colleagues. They have been my family in Melbourne, in no particular order: Xiaoyu Sun (Doctor to be), Yanjie Zhao (Doctor to be), Jiawei Wang (Doctor to be), Mingyi Zhou, Yue Liu, Yonghui Liu, Haowei Quan, Biru Zhang, Dr. Xiao Chen, Dr. Jianmei Hou, Dr. Changyuan Hu.

Finally, my deepest gratitude goes to my parents for their invaluable love and to my sisters for their unconditional support. It is the encouragement and inspiration from them that save me from the hardest time. Without support from them, I would never ever become what I am now.

Contents

List of Figures

List of Tables

Abbreviations

Acronym	**W**hat (it) **S**tands **F**or

Glossary of Terms

Chapter 1

Introduction

1.1 Background

Android Platform. Nowadays, mobile devices have become affordable and ubiquitous. They ease users' daily lives with many and varied mobile applications that provide functions including, but not limited to, online shopping, streaming content, games, banking, transport, education, health, and keeping in touch with friends and family. The Android Operating System (OS) is currently the most adopted OS on all available mobile devices, occupying more than 87% market share [1], with over 6.5 billion customers spanning more than 190 countries [2, 3]. The modern mobile application development model enables developers to quickly implement their function intentions by providing a Software Development Kit (SDK), which offers a set of development tools, including debuggers, device emulators, APIs, documentation, and tutorials, instead of building applications from scratch. With the help of the modern development model, more than 3.81 million applications (apps) are published by over 1.16 million publishers on Google Play Store, the official app release site. In addition, Application Programming Interfaces (API) are also distributed with Android SDK empowering developers to access Android stack functionalities on customers' mobile devices, ranging from inter-component communication [4] to hardware interactions.

Compatibility. In the context of Information Technology (IT), the ability of software and devices from different sources to work together without requiring modification is referred to as *compatibility*. Conventionally, this term includes both hardware compatibility and

software compatibility. Hardware compatibility refers to the smooth interaction between different hardware components with different CPU architectures, buses, motherboards, etc., while software compatibility involves software running smoothly on diverse hardware and/or various operating systems. Keeping the compatibility of both hardware and software is important as it ensures the optimal user experience for device and software customers. If compatibility is successfully maintained between different hardware and software, users are free to operate under different contexts while having the same app user experience.

Android is designed to function across various devices, encompassing phones, tablets, and even televisions. This diversity offers an app developer an extensive potential user base. To ensure app success over all these different devices, apps should be adaptable, accommodating feature differences and offering a versatile user interface that adjusts to various screen setups. In the context of Android development, the term *compatibility* refers to two types of compatibility: *device compatibility* and *app compatibility* [5, 6]. Device compatibility points to the device that could accurately execute apps developed for the Android execution environment, which are regulated by the Android compatibility program [7]. While an app is considered "Android compatible" when it is compatible with a diverse array of device configurations as some features may not be available on all devices.

Since the first release of Android OS in 2008, 33 Android versions have been published, each with several identifiers to uniquely represent the version. These include Android version numbers (e.g., Android 8.0.0), Android codenames (e.g., Tiramisu) [8] [1], Android version codes (e.g., LOLLIPOP_MR1), and Android SDK/API levels (e.g., API level 30) [9]. In total, there are 33 Android levels, ranging from API level 1 to the latest API level 33. Except for the API level 20 which was released as the first version of Android wear, the remaining are all published as the underlying OS for smartphones and tablets. Even though the rapid releases of Android along with its SDK facilitate the development of Android apps, they still pose some barriers to app development, such as compatibility issues. Compatibility issues are encountered with error messages such as "no such API method" or runtime crashes when released apps run on some Android versions but were developed and tested on other versions.

[1] Google has dropped the usage of codenames from Android 10. https://blog.google/products/android/evolving-android-brand/

Listing 1.1 demonstrates an example of typical app compatibility issues. The code snippet was originally reported in [10]. The API *startDrag()* called on Line 7 was introduced into SDK after level 11. However, the minSDKVersion of this app is set to 10. The term `minSDKVersion` is used by app developers to configure the minimum API level of the underlying Android OS in which the app could be installed, including higher API levels. If the app is installed on devices with underlying OS API level 10 and calls the method `startDrag()` without the proper "if-else" guard check, a "NoSuchMethodError" exception will be thrown, which may lead to a crash when running on such devices. The statements beginning with the + signs indicate possible fixes for the potential incompatibility problem. The method `startDragAndDrop` was introduced from API level 24. It is necessary to check the version of the underlying OS before invoking the API to ensure the seamless running of the app on all Android releases (from the regulated `minSDKVersion` to the latest release).

```
1  public class MainActivity extends Activity{
2    private TextView mView;
3    protected void onCreate(Bundle bundle) { ...
4 +    if(Build.VERSION.SDK_INT >= 24)
5 +      wrapper(mView, c, s, null, i);
6 +    else
7        mView.startDrag(c, s, null, i);
8    }
9 +  private wrapper(View v, ClipData c, ...) {
10 +   v.startDragAndDrop(c, s, o, i);
11 + }
12 }
```

LISTING 1.1: Code examples.

1.2 Motivations and Objectives

Thesis Research Motivations. Due to the rapid evolution of the Android OS along with its APIs and various customization releases over the last decade, developers need to take great care to handle compatibility issues. In general, compatibility issues could be caused by two different API provision approaches, including different APIs (i.e., APIs with different signatures) and the same API signature but different implementations

given by different Android versions or customizations. Even though some approaches were proposed to discuss the causes of the occurrence of compatibility issues and even further to identify these issues, they almost all focus on the compatibility issues induced by different API provisions in the official consecutive Android releases. To fill the research gap, we extend the current research by not only discussing the issues caused by the API having the same API signature but different implementations supplied in different Android versions but also APIs among different Android customizations. Going one step further, we also propose a systematic approach to pinpoint special API supplies to advance the abilities of existing issue detection approaches.

The APIs are typically added or deleted in newer releases and offered as specific supplies in certain Android releases. Android app developers rely on the documentation [11] for APIs to access the underlying OS functions and hardware resources. They are not only required to check the availability of these APIs in the documentation but also to ensure behavioral consistency for APIs provided in the continuous Android releases. However, app users would still experience failures if the API specifications are not correctly and promptly updated in the documentation by the OS maintainers. Inconsistencies between the API's implementation updates and changes in its specifications arise with the evolution of Android. Although previous studies have investigated various aspects associated with the evolution of the Android API [10, 12–16], there has been no research on issues in the documentation of behavioral changes in the platform APIs. Additionally, no study has identified the extent to which these issues may affect real published Android apps.

The popularity of Android is supported by its open-source nature, enabling device vendors and OS providers to satisfy a wide range of customers' special needs with their own Android OS customizations. To build OS customizations, device vendors and third-party OS providers modify their OS copies cloned from the official Android OS releases, which are available in the repositories managed by the Android Open-Source Project (AOSP) [17]. When creating new customized OS versions, the customizations' maintainers always add new features, modify, or even delete existing ones. These changes may lead to differences compared to the original code of the OS, resulting in divergent versions of Android OS. Moreover, these customizations' developers also need to merge updates periodically from the official releases to include bug fixes, security patches, and new functionalities [15, 16, 18], etc. Unfortunately, such merge

operations can span the changes characterizing the customized OSs, leading to merging conflicts. These conflicts-involved methods can introduce differences between the original APIs and the customized methods provided in the customizations. Android apps invoking such APIs and running on the customized OSs would unexpectedly experience a range of compatibility issues (i.e., issues that prevent apps from working as expected) [10, 14, 19–22]. Therefore, further research is required to characterize conflict-related methods and help app developers understand issues induced by such methods.

Compatibility issues are considered one of the most severe problems in the Android ecosystem. On the one hand, they can cause unsuccessful installations or crash at runtime if installed successfully, negatively impacting the user experience. On the other hand, they increase the difficulty of app development. The vast number of device-Android version combinations increases technical complexities exponentially for app developers, who must take into account a dizzying number of devices and OS versions. To address these issues, many approaches have been proposed to analyze the compatibility issues of Android apps. For example, Li et al. [14] designed and implemented a prototype tool called CiD that mines the evolution of the official Android framework codebase to locate evolution-induced incompatible Android APIs, i.e., new methods introduced in or existing methods being removed from the latest framework versions. And then it detects compatibility issues induced by these identified APIs. Wei et al. [23] proposed a prototype tool called Pivot for characterizing device-specific incompatible APIs, e.g., APIs that are available for certain devices but not for others. However, it is still unclear what the status quo of Android app compatibility analysis is, what their strengths and weaknesses are, and to what extent they can identify all the possible incompatible Android APIs and their induced compatibility issues in real-world Android apps. Furthermore, it is also unknown to what extent can we reproduce their experimental results and how well do the tools compare with each other in terms of gathering incompatible APIs and detecting compatibility issues.

Objectives. In summary, compatibility issues are widespread among Android apps and have proven to be one of the most severe problems in Android. Although several automatic approaches have been proposed by fellow researchers in recent years to identify compatibility issues, they almost all focus on evolution-induced (the addition and deletion of APIs as the evolution of Android OS) compatibility issues. However,

little attention has been paid to the implementation of issue-inducing methods, which have the same signatures but different implementations among different official Android releases and third-party Android OS customizations. Furthermore, the differences between the proposed detection tools have not been extensively investigated. To address these research gaps, we want to answer three key research questions that we have identified, namely:

- **Research Question 1: What is the status of silently evolved methods among official Android OS releases?** With the release of the new version of Android, incompatible APIs (i.e., newly added and deleted APIs) are introduced to app developers, adding the burden of ensuring that apps implemented with incompatible APIs run seamlessly on different Android OS versions. Silently evolved methods (SEMs) in official Android OS releases refer to APIs with continuous supply but whose behaviors have been changed, increasing the difficulty of app development. However, the existence of SEMs in Android has been overlooked by researchers and practitioners. By investigating SEMs in Android, OS maintainers are advised to take more attention to the consistencies of the continuous API supply. Consequently, app developers would have higher confidence in implementing their app with these continuous API supplies, and app users would receive a consistent user experience on different OS versions.

- **Research Question 2: How do customized Android OSs evolve with respect to the official Android OS releases?** The open-source nature of Android has led to numerous customized OS distributions in the wild. These OS customizations are required to integrate bug fixes, security patches, and new features from official OS releases to provide consistent underlying support for app execution. However, merge conflicts can occur when developers of customized OS attempt to merge updates from official releases, leading to API inconsistencies between the official release and customization supply. These inconsistencies make it challenging to ensure seamless app execution on different OS customizations. Through an examination of the evolution of customized OS, another type of inconsistent API has been discovered, highlighting the need for developers to take extra effort to provide consistent APIs when resolving merge conflicts.

- **Research Question 3: What are the differences between the state-of-the-art approaches in compatibility issues identification?** To address the severe problem of compatibility issues (induced by the invocation of the aforementioned inconsistent APIs without proper guard checks) in Android apps, different approaches have been proposed to pinpoint such issues in apps. However, researchers and app developers still lack a comprehensive understanding of compatibility issue analysis in Android apps. By replicating and comparing detection approaches collected from the literature, the advantages and weaknesses of the detection tools would be concluded. App developers can then choose the appropriate detection tools to identify compatibility issues in their developed apps according to their special detection requirements. Meanwhile, researchers can be inspired by the weaknesses of the detection tools to propose more advanced and efficient approaches to overcome the limitations.

1.3 Thesis Scope

This thesis focuses on a severe problem for many Android apps, developers and users: compatibility issues. With the rapid evolution of the Android Operating System, not only the addition and deletion of Android APIs but also other complex changes in Android APIs can induce compatibility issues. To address these issues and identify compatibility problems, we developed automatic approaches and conducted extensive studies to answer the research questions mentioned earlier.

- **AndroSea** - An automatic approach to identify silently-evolved methods from official Android releases.
- **Empirical Study** - An extensive study to investigate the evolution of third-party Android OS customizations and their conflict-related methods.
- **Replicability Study** - A replicability study to compare the approaches in issue detection.

AndroSea (Chapter 3). App developers heavily rely on the provided APIs to develop their Android apps. To avoid compatibility issues (i.e., issues that prevent apps from

working as expected when running on different versions of the API) in their Apps, they always take extra efforts to continuously monitor API changes on the official development site. Despite a large number of studies on compatibility issues in the Android API, the research community has not yet investigated issues related to silently-evolved methods (SEMs). These methods are functions whose behavior might have changed, but the corresponding documentation did not update accordingly, which would mislead developers developing Android apps with unexpected failures emerging when running the apps. To shed light on this type of issue, we proposed the automatic approach ANDROSEA to identify and characterize such SEMs across ten consecutive versions of Android releases. It first pre-processes the Android code base by selecting API-related modules with a manually constructed whitelist, as some modules are test-related and command-tools implementations, which are irrelevant to development APIs. Afterward, it extracts the API signature along with its implementation from the selected API-related modules. By comparing API signature and its implementation between two continuous Android releases across ten consecutive versions of Android releases, the SEMs (Silently Evolved Methods) are successfully spotted. The consequences of calling these SEMs are also evaluated among a randomly curated 1,000 Android apps.

- *This work has led to a research paper — **Identifying and Characterizing Silently-Evolved Methods in the Android API**, which has been published at The 43rd ACM/IEEE International Conference on Software Engineering, SEIP Track. (ICSE-SEIP 2021), 2021.*

Empirical Study (Chapter 4). To satisfy the diverse needs of customers, device vendors and OS providers always rely on the open-source nature of the Android OS and offer customized versions of the Android OS. When a new version of the Android OS is released, device vendors and OS providers need to merge the changes from the Android OS into their customizations to account for bug fixes, security patches, and new features. Because developers of customized OSs might have made changes to code locations that were also modified by the developers of the Android OS, the merge task can be characterized by conflicts, which can be time-consuming and error-prone to resolve. To resolve the methods-related conflicts, customized OS maintainers could introduce inconsistencies compared to the official methods, which could eventually result in compatibility issues in Android apps. Aside from the incompatibilities induced

by the addition and deletion of APIs, as well as the introduction of silently evolved methods with the evolution of the official Android releases, the API inconsistencies resulting from the merge conflicts in Android customizations are also widespread in the Android ecosystem. Therefore, we investigate another type of API inconsistency in this research. To provide more insight into this critical aspect of the Android ecosystem, we analyze how often the developers from the customized OSs merge changes from the Android OS, how often they experience merge conflicts, and the characteristics of these conflicts and their serious ramifications on real Android apps.

- *This work has led to a research paper —* ***Do Customized Android Frameworks Keep Pace with Android?****, which has been published at The 19th International Conference on Mining Software Repositories (MSR 2022), 2022.*

Replicability Study (Chapter 5). The severe problem of compatibility issues is mainly caused by the fast evolution of the system itself and the various customizations independently maintained by different smartphone manufacturers and third-party OS providers. In detail, the fast evolution of the Android OS and the extensive adoption of OS customizations have brought Android app developers a plethora of inconsistent APIs that are proprietary for specific underlying OSs, such as the newly added and deleted APIs, as well as silently evolved methods in newer official Android releases, the resolution of merge conflict related APIs in OS customizations, etc. Compatibility issues refer to the invocations of inconsistent APIs without proper guard checks during app development. Many efforts have attempted to mitigate their impact via approaches to automatically pinpoint compatibility issues in Android apps. Unfortunately, at this stage, it is still unknown if this objective has been fulfilled, and if the existing approaches can indeed be replicated and reliably leveraged to pinpoint compatibility issues in the wild. We, therefore, first conduct a literature review on this topic to identify all available approaches. After that, we go one step further to empirically compare those approaches and summarize their typical steps in issue detection. Two separate steps, including incompatible APIs gathering and compatibility issues (induced by such incompatible APIs) determination, are commonly leveraged in the nine identified detection approaches. We try our best to reproduce and compare them based on their original and newly introduced datasets with real-world apps containing compatibility

issues. To boost the performance of the existing detection tools and guide future research in the area of issue detection, we propose *AndroMevol* to systematically gather incompatible APIs from six OS providers.

- *This work has led to a paper submission —* ***Automatically Detecting Incompatible Android APIs****, which has been submitted to ACM Transactions on Software Engineering and Methodology (TOSEM), 2022, under Major Revision.*

1.4 Thesis Contributions

This section summarizes the main contributions of our research in the thesis in which our past published papers are included.

Chapter 3 introduces the automatic approach, AndroSea, to identify and characterize SEMs across ten continuous Android API levels. In this work we make the following key contributions:

- A characterization of SEMs across the ten most used versions (at the time we started our study) of the Android API. We analyzed method updates across ten versions of the Android API and manually confirmed that 363 methods out of a sample of 562 publicly-accessible methods contain semantic changes. We also characterized the nature of SEMs and found that developers might have accidentally introduced the majority of them as most of them evolved only once.
- A study of how SEMs impact Android apps. We investigated whether and how Android apps use SEMs in a sample of 1,000 real-world Android apps. We found that SEMs are commonly used by Android apps, and such usage could indeed lead to compatibility issues as app developers have added checks to prevent the execution of SEMs on certain versions of the Android API.
- A tool to identify SEMs and the experimental data containing the findings of our study. To identify and characterize SEMs, we designed a technique and implemented the approach in a prototype tool called AndroSea. AndroSea identifies methods across two versions of the Android API that have the same signature, and the same method comment, but different method body. The tool and

the experimental data are publicly available at https://github.com/MobileSE/AndroSea.

Chapter 4 presents an extensive empirical study to investigate the evolution of customized Android OSs, especially the merge conflicts that arise from incorporating bug fixes, security patches, and other changes from official Android OS releases. In this study, we make the following contributions:

- An empirical study that analyzes the version control history of eight open-source customizations of the Android OS and investigates how their developers merged changes from the new releases of the Android OS into their projects. The study results show that OS customization maintainers, on convention, take more than 20 days to bring changes from the official Android releases and they perform these merge operations only for 9.7% of the released versions of the Android OS. The OS maintainers will encounter at least one conflict in 41.3% of the merge operations, with 58.1% of the conflicts requiring them to take actions to update the customized OSs. Furthermore, in over 40% of cases, developers disregarded the portion of the conflict from the official Android repository, potentially leading to the introduction of compatibility issues.
- A quantitative analysis of the methods affected by merge conflicts in randomly selected 1,000 Android apps. The experimental results show that 644 (or 64.4%) of 1,000 apps accessed conflict-affected methods. The fact that over half of the randomly selected apps access such methods demonstrates that the potential impact of customization changes could be huge and the compatibility issues in Android apps could be aggravated by the popularity of the customized Android OSs.

Chapter 5 discusses differences between existing issue detection tools and the new approach in incompatible APIs systematic collection. In this work, we make the following key contributions:

- A systematic literature review was conducted, which successfully identified nine primary publications that have proposed automated approaches to characterizing Android app compatibility issues. After careful analysis, we summarized five

identified types of API-induced compatibility issues: Evolution-induced (Method), Evolution-induced (Field), Device-specific (Method), Device-specific (Field), and Override/Callback. Unfortunately, none of the existing approaches can tackle all five types of API-induced compatibility issues. The most recent one, ACID [24], can only handle three out of the aforementioned five types.

- A replicability study was adopted to replicate the selected detection tools against their original experimental dataset. Our experimental results show that the majority of experimental results could indeed be reproduced. The remaining small number of inconsistent results (yielded by IctApiFinder and FicFinder) was mainly caused by unnecessary updates of the tools (such as dependency fixes) and apps (due to the unrecorded versions of the apps).

- A comparative fair comparison was launched w.r.t. two common sets of benchmark apps. Experimental results show that (1) compatibility issues detection approaches that achieved their purpose via systematically harvested incompatible API rules (such as CiD and IctApiFinder) could identify significantly more issues than those that had their rules summarized manually, and (2) the intersection among the results reported by the selected tools was relatively small. These detection tools still have limitations to achieve sound compatibility issues detection. Furthermore, it is essential to leverage systematic approaches to mine incompatible APIs to advance the detection of compatibility issues.

- A systematic approach to harvesting incompatible APIs, including methods and fields, among five popular OS brands, including Huawei, Xiaomi, Oneplus, OPPO, and Samsung alongside AOSP. We proposed a new prototype tool called *AndroMevol* to harvest incompatible APIs from the official Android frameworks and 52 customized Android frameworks extracted from the released ROMs of the five popular brands. With the *AndroMevol*, we could collect a total of 397,678 incompatible APIs including 195,883 previously untouched ones, which could be accessed by Android apps inducing compatibility issues. Compared to other API harvest tools, *AndroMevol* performs better by harvesting at least eight times more incompatible APIs.

In summary, this Ph.D. thesis focuses on compatibility issues' introduction and detection. In addition to the typical compatibility issues induced by the invocations of APIs

(added or deleted in newer Android releases) without proper guard checks, we try to identify different types of APIs that can still induce compatibility issues. We identified that not only the API with the same signature but different implementation in different API levels among official Android releases but also the API with the same signature and different implementation provided in third-party Android OS customizations might also result in compatibility issues among Android apps. Going one step further, we also investigated the status quo of compatibility issue detection in the literature. With an extensive systematic literature review and replicability study, we successfully identified nine relevant research work focusing on issue detection but we could only replicate four of them. Our replication study w.r.t. two common experimental datasets shows that some detection tools perform better than others and the limitation of incompatible APIs gathering exists in the process of issue detection among the existing detection tools. To overcome the API harvest limitation, we proposed *AndroMevol* to systematically collect incompatible APIs from the official and 52 customized Android frameworks that are extracted from the released ROMs of five popular brands.

1.5 Thesis Structure

In this thesis, Chapter 1 introduces the introduction and motivations of the thesis. Chapter 2 summarizes the related literature which includes different compatibility issues detection tools, such as CiD, ACID, etc. Chapter 3 introduces the automatic approach, AnDroSea, to identify and characterize SEMs across ten continuous versions of Android. In Chapter 4, we present an extensive empirical study to investigate the evolution of customized Android OSs, especially its merge conflicts due to involving bug fixes, security patches, etc., from the official Android OS releases. In Chapter 5, we discuss the differences among the proposed issue detection tools and propose *AndroMevol* to systematically harvest incompatible APIs. We finally conclude our research in Chapter 6 and discuss future research directions in the same chapter.

Chapter 2

Literature Review

In this chapter, we provide readers with the necessary background [1] to understand current research on characterizing and detecting compatibility issues among Android apps. Compatibility issues in Android apps are typically triggered by invocations of APIs that are available on some Android releases but not on others, without appropriate guard checks. Moreover, compatibility issues could also be caused by invocations of APIs with the same signatures but different implementations among different API levels or customizations. In this Ph.D. thesis, we not only explore the characterization of compatibility issues but also investigate the detection of compatibility issues. We classify the relevant research into three main categories based on our main research questions, which include investigating official Android APIs, exploring of the evolution of third-party Android OS customizations, and detecting of compatibility issues.

2.1 Official Android APIs Investigation

The development of Android apps heavily relies on Android APIs to take advantage of functions and features provided by the underlying OS. However, Android has evolved at a high speed in the last decade, releasing more than 30 versions. App developers not only need to examine the provided API documentation [11] to guarantee correct invocations of APIs, but also adapt their apps to the rapid releases. To relieve such

[1] We include recent publications and provide a comprehensive literature review as the related work in the following chapters cannot be allowed to revise.

a heavy development burden, plenty of research has been conducted by a variety of researchers.

McDonnell et al. [12] carried out a thorough scientific investigation to examine how the evolving APIs have impacted Android apps. By mining the version history data on Github, they determined that Android is indeed evolving rapidly with an average of 115 API updates per month. Even though app development heavily relies on the fast-evolving APIs, app developers are reluctant to adapt their app at the same pace, and only about 22% outdated APIs are eventually replaced with updated ones 14 months later on average. Furthermore, they confirmed that the APIs adopted by app developers positively correlate with the APIs Google update most frequently. Li et al. [15] proposed a prototype called CDA to characterize deprecated APIs in Android framework code bases. Their experimental results showed that there are inconsistencies between API annotations and documentation, which would induce severe consequences in apps. It is popular libraries that contribute to the majority of invocations of deprecated APIs, even though library developers are more likely to update deprecated APIs than app developers. In addition, alternative APIs are always provided when the APIs are annotated as deprecated. However, app developers are reluctant to replace the deprecated APIs with alternatives in their app development.

Linares-Vásquez et al. [25] first conducted an empirical analysis to determine the relationship between the success of 7,097 Android apps and their underlying supportive Android APIs. Their experimental results show that successful apps are prone to adopt significantly less fault-prone and change-prone APIs. Linares-Vásquez et al. [26] also explored the reactions on Stack Overflow when Android APIs are updated. Their investigation shows that developers will have much more discussion when Android APIs are changed, especially when they are deleted. It is necessary to inform developers timely when sensitive updates are introduced in APIs in new releases. Moreover, Linares-Vásquez et al. [27] presented a quantitative and qualitative analysis of the energy-greedy APIs in 55 free Android apps. Their study unveiled that a rational database working as a persistent layer for apps might have a huge impact on energy consumption. Bavota et al. [13] did a similar evaluation on the relationship between the success of 5,848 free Android apps and the utilization of fault- and change-proneness of APIs in them. They further conducted a survey with 45 Android developers to provide a quantitative and qualitative explanation of the correlation findings. The survey confirmed the

observations that successful apps are all prone to invoke less fault- and change-prone APIs even though avoiding such fault- and change-prone APIs during their development is non-trivial.

The success of Android apps has attracted attackers who release malicious apps to steal privacy-sensitive data. To distinguish malicious apps from benign ones, Chan et al. [28] proposed a static mechanism for malware detection that considers API calls. Compared to other approaches that only take permissions into account, their approach could achieve a high accuracy rate of more than 90%, showing that the information of API calls is helpful in malware identification. Wu et al. [29] took calls of dataflow-related APIs into account when adopting KNN classification model to spot malware. They computed a malicious weight value for each selected dataflow-related API during privacy leakage analysis in malware identification. Experiments on 1,160 benign and 1,050 malicious samples revealed that dataflow-related APIs are helpful in pinpointing Android malware. Tao et al. [30] also proposed an automatic malware detection system, MalPat, which tackles the malware detection problem from an API perspective. They extracted highly sensitive APIs via API usage comparison between malicious and benign Android apps. Experimental results show that MatPat achieved better results in both precision and recall compared to other state-of-the-art approaches in malware identification.

In addition to the public Android APIs, there are some APIs that cannot be accessed conventionally but with unconventional approaches (e.g., reflection), which are referred to as inaccessible APIs. Li et al. [16] did an empirical study on 17 important releases of the Android framework source code base and found that the inaccessible APIs are widely implemented in Android frameworks. However, they are actually provided in an unstable approach as some of them would be removed in future releases. With regard to the adoption of inaccessible APIs in Android apps, the experimental results reveal that many Android apps indeed access such APIs. App developers seem to not take the risk of removals of such APIs into account but are only intrigued by the functions provided by them when developing their apps. The Google Play store either does not provide an approach to detect the usage of inaccessible APIs. Yang et al. [31] conducted a systematic study to investigate the use and design of Android non-SDK APIs, which also require unconventional approaches to invoke. Google has strengthened the restriction to access non-SDK APIs since Android 9 so as to enhance the stability of

Android apps. However, their experimental results show that such APIs are still used by developers via unusual approaches, such as double-reflection and call stack breaking, because they are critical to special feature implementations in app development. Moreover, they are abused by malicious app developers to implement malicious operations. He et al. [32] focused on the vulnerability aspect of the non-SDK APIs. To investigate the impacts of the vulnerability of such non-SDK API, they proposed an automatic approach to analyze and compare the security mechanisms in various official Android releases. They eventually found 32 serious vulnerabilities that could be used to trigger attacks in Android 6. In the most recent release at the time of their research study, only 25 APIs are identified as vulnerabilities in Android 12 but no harm has been spotted. The results show that the inconsistent security enforcement added by Google has successfully addressed the vulnerabilities in non-SDK APIs.

To help app developers upgrade their apps with newly released APIs, Fazzini et al. [33] proposed a technique called AppEvolve, which involves four steps to automatically incorporate new updates into apps. AppEvolve extracts examples of app updates and creates update patches from existing code bases. Experiments on 15 real-world apps reveal that AppEvolve was able to update 85% of the API usages and validate 68% of these updates automatically, demonstrating its usefulness. Thung et al. [34] replicated AppEvolve and determined whether it could be generalized to more cases. By extending the experimental data to other mobile apps with the same API calls, the results showed that 81% of the curated mobile apps cannot be correctly updated. They further categorized the limitations of AppEvolve and proposed remedial approaches to advance it. Finally, they achieved better results and also pointed out some future research directions, such as code normalization. Haryono et al. [35–37] first proposed CocciEvolve and then published an advanced version AndroEvolve to overcome the limitations of CocciEvolve through data flow analysis and variable name denormalization. Experiments on 360 target files showed that AndroEvolve could achieve more than 26.90% correct and readable updates compared to CocciEvolve.

Developers of Android applications heavily rely on documentation [11] to access OS functions and hardware resources through APIs. They are not only responsible for checking API availability and ensuring consistent behavior across Android releases but also for addressing potential discrepancies between API implementation updates and specification changes. Failures in updating API specifications by OS maintainers can

lead to user experience issues. While prior studies have explored Android API evolution, none have focused on behavioral change documentation issues or their impact on real-world Android apps.

2.2 Evolution of Third-party Android OS Customizations

The open-source nature of Android enables third-party OS providers and device vendors to add unique features to their customizations, promoting the popularity of Android OS among mobile customers. The vast number of customizations in the Android ecosystem has gained the attention of plenty of researchers. Wu et al. [38] proposed a SEFA analysis framework to evaluate the security impact of vendor customizations on Android devices. They focused on the pre-loaded apps and explored their provenance, permission usage, and vulnerability distribution. Experiments on ten customization releases, including five different vendors and each containing two models, showed that vendors' modifications contributed to more than 60% vulnerabilities in the released firmwares, and over 50% pre-loaded apps were published with over-privileged issues. Zhou et al. [39] paid attention to the security risks in the process of customizing Android for different device drivers, such as cameras, GPS, NFC, etc. They first built a dynamic analysis tool, ADDICTED, performing dynamic analysis to relate the operations on a security-sensitive device to its relevant Linux files, and then examined if these files were under-protected on the Linux layer compared with the counterpart on the official Android OS. If there is a discrepancy in Linux files between customized Android OS and their counterpart in the official Android OS, and the customized ones extended the operation privileges, it is sufficient to conclude that security risks emerge. Experiments conducted with ADDICTED on popular smartphone models unveiled that critical flaws were widespread among phone models and could even be exploited by attackers to take pictures, screenshots, and record users' input. Gallo et al. [40] analyzed the impact on security architectures due to the Android OS customization. They collected five different distributions, including Google Nexus 4, Google Nexus 5, Sony Z1, Samsung Galaxy S4, and Samsung Galaxy S5, in which OS customizations based on Android 4 were deployed. Their analysis concluded that security concerns were aggravated by such customizations, such as attack surface expansion, out-of-order permission control, hard-to-maintain compatibility, etc., and security-sensitive users should

avoid such heavily customized distributions. Aafer et al. [41] conducted a large-scale differential analysis on 591 customized Android OSs regarding the selected security features to examine the security problems introduced by these customized images. Their experiments showed that the discrepancies in the curated OS customizations were widespread and could indeed lead to severe security breaches. Shao et al. [42] further proposed a static analysis tool, Kratos, to unveil inconsistencies in security enforcement of Android OS, including customized ones. Experiments on four versions of the official Android and two OS customizations discovered 14 highly-exploitable vulnerabilities and confirmed that systematic problems exist in Android.

Vulnerabilities associated with OEM commands are found common and could induce severe consequences, such as data exfiltration. To avoid future vulnerabilities and strengthen the robustness of Android releases, Hay et al. [43] presented an analysis of the Android fastboot interface and proposed a tool to pinpoint vulnerabilities associated with OEM commands from multiple devices, including Motorola, OnePlus 3/3T. They also provided several mitigation techniques and suggested OEM developers pay more attention to the potential vulnerabilities. Iannillo et al. [44] and Cotroneo et al. [45] released a novel “gray-box” fuzzing tool, Chizpurfle, for customized-specific Android services. It leverages the technique of dynamic binary instrumentation to determine test coverage and guide fuzz input generation. The experimental results on the Samsung proprietary OSs showed that it was valid and effective in bug detection. Farhang et al. [46] conducted a comprehensive study on the Android security bulletins collected from AOSP, Samsung, LG, and Huawei. With the collected total of 3,171 unique CVEs, they concluded that vendors adopt different structures for vulnerability publishing, vendors tackle Qualcomm-related CVEs differently compared to CVEs from other layers, and vendors seem to handle CVEs with Android Git repository references timely. To disentangle vendors’ customizations from Android Open Source Project (AOSP), Google has published many requirements, lots of automatic routines, and Project Treble to determine the compliance for AOSP. Possemato et al. [47] conducted a large-scale study to investigate whether these requirements are satisfied or not and whether the customizations are in compliance with AOSP. Investigations on four main components, including SELinux configurations, system binaries hardening, init scripts, and the Android Linux kernel, presented that 20% experimental collected ROMs violated at least one requirement, even 11 of them are customized by Google itself. The problems are not

alleviated with recent new releases. Yu et al. [48] worked on the integrity of SEAndroid policy rules. To provide customized features, ROM developers always publish their own new rules to meet the functionality extensions. However, such newly added policy rules could breach the official defense in SEAndroid. The authors presented a tool, SEPAL, to automatically extract policy rules and determine if these customized rules are regulated or not. Their experimental results showed that the approach was effective by providing a 15% accuracy rate on average and identified 7,111 unregulated policy rules with low false positives, including some known policy issues. The widespread policy rule issues in customized Android ROMs suggested that new improvements are necessary for robust customizations. Hou et al. [49] presented a large-scale comprehensive analysis of Android firmware security based on 6,261 Android images from 153 vendors and 602 Android-related CVEs. They proposed an analysis framework, AndScanner, to automatically crawl and parse ROM images and analyze the pre-installed apps and security patch information. Their analysis concluded that the security patch delay was common in Android OS images, even for popular ones, such as Huawei, Samsung, and the image maintainers were keen to involve new features in their customizations rather than fix potential security issues. Even if the image developers claim to have completed security patches, there are still patches missing. One possible reason behind this is that sufficient testing before the image release requires quite a lot of effort, which is difficult to achieve due to a shortage of developers. Moreover, security risks are widespread among pre-installed apps. Image developers need to put in great effort to test both the images themselves and the pre-installed apps to guarantee a robust OS release.

In addition to analyzing security issues in customized OS releases, researchers should also focus on the implementation of these customizations, particularly the process of involving updates from AOSP. To involve updates from AOSP, customized OS developers need to execute Git merge operations, which can result in merge conflicts. There is a significant amount of research work concentrating on merge conflicts. Mens [50] studied different merge techniques and compared them with respect to a number of metrics, including expressiveness, formality, granularity, domain independence, customizability, scalability, efficiency, and accuracy. He concluded that the three-way operation-based merge technique is a potent approach that takes into account not only textual conflicts but also syntactic and semantic conflicts. Most of the commercial merge tools

spend their efforts on textual-based merge, which only takes effect in terms of simple merge conflicts and is inadequate in many situations. The promising merge approach would be the graph-based one, which represents the underlying software as a graph. In general, more research is necessary, regardless of whether it is syntax-based, semantics-based, or simple textual-based, to provide a more powerful merge technique. Mahmoudi et al. [51] focused on merging operations in customized Android Operating System and proposed a toolchain to explore the changes and overlaps between customized OS and AOSP. To carry out the study, they selected eight versions of the open-source Android customization, LineageOS. Conventionally, vendor OS developers need to merge updates from new release Android versions or re-apply their customizations on the new OS releases to provide a compatible customized OS. In the study, they thoroughly investigated the changes and overlaps in source code between LineageOS and AOSP. The empirical study of the selected LineageOS showed that both LineageOS and AOSP update similar parts, and the source code related to settings and user interface was updated by both LineageOS and AOSP maintainers. More importantly, they unveiled the possibility of completing merge operations automatically with new auto-merge tools in the future. Sung et al. [52] carried out the first industrial case study of merge conflicts in divergent forks based on Microsoft Edge and classified the merge conflicts into three different types (i.e., textual conflicts, build breaks, and test failures). To help developers automatically fix merge conflicts related to build breaks, they proposed an auto-merge tool, MrgBldBrkFixer, and evaluated it in real Edge divergent (i.e., Microsoft Edge Beta). Experiments on three months development data showed that MrgBldBrkFixer could successfully generate patches for 40% of the merge conflicts. Ellism et al. [53] compared two refactoring-aware merging techniques, RefMerge implemented based on Git and with refactoring considered and IntelliMerge developed as a graph-based while refactoring aware technique, as refactorings are always involved in merge conflicts. Experiments on 2,001 merge scenarios from 20 open-source projects revealed that RefMerge could successfully resolve conflicts in 25% merge scenarios and induce conflicting LOC in 11% scenarios while IntelliMerge could resolve 24% conflicts and result in conflicting LOC in 30% merge scenarios. They concluded that IntelliMerge performs better under the situation with ordering and formatting conflicts while RefMerge does well with conflicts in which a number of refactorings are involved.

When crafting new customized OS iterations, maintainers of these customizations introduce new features, modify existing ones, and sometimes even remove features. Such modifications can result in deviations from the original OS code, leading to divergent Android OS versions. Additionally, developers of these customizations need to periodically integrate updates from official releases, which include bug fixes, security patches, and new functionalities [15, 16, 18]. Unfortunately, these integration processes can encounter conflicts that arise from the unique changes in customized OSs, causing conflicts during merging. These conflicts can result in differences between original APIs and customized methods provided in the customizations. As a consequence, Android apps that utilize such APIs and run on customized OSs might unexpectedly encounter compatibility issues, disrupting their expected behavior, which has never been touched by researchers.

2.3 Compatibility Issue Detection

The rapid release of Android and Android customizations provided by vendors and third-party OS developers imposed a severe problem, fragmentation, in the Android ecosystem. App developers are required to take extra efforts to guarantee seamless running on different devices and underlying OS versions. To understand such a situation in the Android ecosystem, plenty of research work has been published. Han et al. [54] analyzed bug reports with approaches of Labeled-LDA (Latent Dirichlet Allocation) and LDA from two famous vendors, HTC and Motorola, and classified topics of the bug reports as common or unique (i.e., vendor-specific and induced by fragmentation). Finally, their analysis results revealed that Labeled-LDA produced more feature-oriented topics than LDA, which reflects how hardware fragmentation affects the bug report.

Khalid et al. [55] focused on game app testing. To fix device-specific issues, developers are required to test their apps on different devices. Therefore, they proposed an approach to mine device information from apps' reviews and found that most of the reviews came from only a small set of devices and the collected device information could also be utilized for new similar apps test. Lu et al. [56] further proposed a technique, PRADA, to prioritize Android device models by mining a large-scale dataset for

app testing. The PRADA technique utilizes a collaborative filtering technique to predict major device models for apps, even for new ones. Experiments on 200 Android apps from the categories of Game and Media confirmed the validity of PRADA and provided useful guidelines for app developers.

Wei et al. [19] is the de-facto first research project to identify compatibility issues in Android apps. They first investigated the symptoms and root causes of compatibility issues and then proposed an issue detection tool, FicFinder, to identify compatibility issues. The FicFinder first collected incompatible APIs (i.e., specific in some Android releases) and then analyzed Android apps through a static code analysis approach. Experiments on 27 open-source Android apps demonstrated that FicFinder was effective in compatibility issue identification as many unknown issues were detected and confirmed by app developers.

Fazzini et al. [57] tried to detect cross-platform inconsistencies in Android apps. They proposed an inconsistency detection tool, DiffDroid, to compare app running behaviors collected from different devices with the same app and the same input and tested on 5 benchmarks over 130 platforms. The experimental results showed that DiffDroid can efficiently pinpoint cross-platform inconsistencies with reasonable false positives. Dilhara et al. [58] focused on issues caused by incompatible permission uses and presented a tool, ARPDroid, to repair such issues automatically. Experimental results demonstrated the effectiveness of ARPDroid that it could finish detection with a high accuracy rate of 100% and repair at an average time cost of about two minutes.

Wu et al. [59, 60] detected inconsistencies between the declared SDK version and the API calls in apps. They found that some apps do not provide the target SDK version or specify it wrongly while some apps in their dataset declared a lower SDK version inducing runtime crashes if running in the designated lower Android versions. Li et al. [61] introduced a compatibility issues detection tool, ELEGANT, to overcome the limitations of the previous detection tools which are imprecise program analysis and incapable of third-party libraries processing. The tool, ELEGANT, can effectively identify compatibility issues while reducing false positives by about 70%.

Li et al. [14] have designed and implemented a prototype tool called CiD that mines the evolution of the official Android framework codebase to locate evolution-induced incompatible Android APIs, i.e., new methods introduced in or existing methods being

removed from the latest framework versions. With the help of the extract incompatible APIs and path-sensitive data-flow analysis, CiD could pinpoint compatibility issues in Android apps. He et al. [10] also provided an issue detection tool, IctAPIFinder, which extracts incompatible APIs from official Android SDKs with additional third-party tools, such as Heros [62], Doop [63], LogicBlock [64, 65], etc. With a different static analysis technique and incompatible API extraction approach, it could also successfully detect compatibility issues in Android apps.

Huang et al. [66] deeply explored the compatibility issues caused by the evolution of callback APIs, and proposed CIDER, which utilizes a graph-based model to detect API callback compatibility issues. Wu et al. [67] conducted a comprehensive survey to summarize the research among Android apps and made a taxonomy of them. They concluded that the usability of Android apps is impacted by fragmentation and it is necessary for researchers to propose valid techniques to fix fragmentation-related issues.

Cai et al. [22] presented a large-scale study to investigate the symptoms and causes of compatibility issues. They classified compatibility issues into two different types, installation time and runtime. To explore the symptoms and causes of issues at installation time, they collected 62,894 apps and concluded that installation time issues always involved minSDKVersion designated in apps. On the other hand, they curated 15,045 Android apps to investigate runtime issues and concluded that runtime issues were widely introduced by the API changes during the update of the underlying Android OS. Zhang et al. [68] investigated the attitude of developers when users perceived compatibility issues regardless of injecting or preventing such issues. Their analysis demonstrated that some developers clearly ignored compatibility recommendations and their intentions are significantly different between malicious apps and benign ones.

Ceccato et al. [69] discussed the utilization of in-vivo testing for Android applications. In detail, they tried to exploit feature models to manage in-vivo testing so as to cover as many configurations, OS versions, and devices as possible. Mobilio et al. [70] focused on backward compatibility issues rising from Android API updates and released a detection tool, FILO, which not only extracts the methods that need to be updated but also suggests key symptoms of execution failure in apps so as to ease issue fix. Villanes et al. [71] introduced an approach, DeSeCT, to facilitate mobile device selection

for app testing. Their experiments showed that the tool could exclude about 15% to 18% unwanted devices saving development costs.

Scalabrino et al. [20, 72] investigated changes in app implementation with respect to API updates and introduced a new tool, ACRYL to not only detect compatibility issues but also to extract potential issue fixes. Their experiments compared to another detection tool CiD concluded that the two approaches are complementary to each other. Xu et al. [73] identified compatibility-related APIs from Android apps with the new proposed tool ICARUS based on the fact that such APIs have biased distribution among code segments. Their evaluation of 10,749 real Android apps with manual validation and case study revealed the effectiveness of the tool.

Guilardi et al. [74] investigated the readiness of Android apps when Google releases newer Android versions. They introduced a Repository Mining Tool to analyze 8,420 open-source repositories and concluded that Android apps would become less ready with the evolution of the Android OS. Mahmud et al. [24] proposed an issue detection tool, ACID, to detect compatibility issues induced by incorrect API calls and callback API invocations (callback APIs are different from common APIs, to some extent). Via static analysis of Android source code and extraction of differences from API releases, ACID performs better than the state-of-the-art approaches by pinpointing issues more accurately while taking less time. Mahmud et al. [75, 76] focused on the impacts of regular updates on Android apps and developed a new approach, APICIA, to analyze the impact. They evaluated APICIA on 219 Android apps and concluded that the API changes could impact 46.30% of tests per app on average and lots of affected statements are not touched by existing tests.

Compatibility issues arise when inconsistent APIs are invoked without appropriate guard checks during app development. Several attempts have been made to reduce the impact of these issues by using methods to automatically identify compatibility problems within Android apps. However, it remains uncertain whether this objective has been achieved, and the feasibility of replicating and effectively utilizing existing approaches to identify real-world compatibility issues remains unconfirmed.

Chapter 3

ANDROSEA: Identifying Silently-Evolved Methods

Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. "Identifying and characterizing silently-evolved methods in the android API." In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 308-317. IEEE, 2021.

3.1 Introduction

Mobile applications (or simply apps) are becoming increasingly prevalent in our lives. For instance, in 2017, US-based users spent an average of two hours and 25 minutes per day using mobile apps. This time accounts for more than 80% of the total time spent by the users on their mobile devices [77]. Android apps, which run on the Android operating system (OS), are the most widely used type of mobile apps and account for over 70% of the mobile OS market share [78]. Android apps are not only extremely popular, but their number is also growing at a staggering speed, with about 35,000 new apps released on Google Play every month [79].

One common trait of Android apps is that they rely heavily on the underlying Android OS, as the OS offers access to a large number of popular and essential app services. Apps can access these services using the application programming interface (API) of

the OS. On average, 25% of all methods and field references in the apps are uses of the Android API [12]. This characteristic facilitates app development [80], but it also creates a tight coupling between the apps and the version of the API used by the apps.

Unfortunately, the Android OS and its API evolve rapidly [12, 13, 25, 81–83], and when a new version of the API is released, app developers need to carefully understand the changes to the API so that they can suitably adapt their apps to also run on the new version of the API. To help app developers in this task, Android provides a curated documentation that app developers can access using the platform website [84]. This documentation is based on the comments associated with the source code of the API, and the comments are created and maintained by the API developers.

When API developers are working on a new version of the API, they not only should create comments that describe newly added API components but they should also update existing comments to report changes in existing API components (e.g., [85]). In the latter task, developers should document both syntactic and behavioral changes introduced in the new version of the API. Although it is important to document both types of changes, it is imperative to document the second class of changes as app developers would otherwise not easily know how to update their apps, and users might experience field failures due to compatibility issues.

Although related work investigated a number of aspects associated with the evolution of the Android API [10, 12–16], to the best of our knowledge, no work has systematically analyzed whether there are issues in the documentation of the behavioral changes in the platform API. Additionally, no study identified the extent to which these issues might affect Android apps. A study on these topics would provide insights for building automated techniques that detect the issues and highlight mitigation strategies against the issues.

To fill this gap, we present an extensive empirical study that identifies and characterizes *silently-evolved methods* (SEMs) in the Android API. A SEM is a method whose implementation is different across two subsequent versions of the Android API, while the method's documentation (i.e., the method's comment) is not changed. In the study, we (i) identify SEMs across different versions of the Android API, (ii) report the characteristics of these methods, and (iii) analyze the impact that SEMs might have on real-world apps. Specifically, we analyzed method updates across ten API releases and manually

classified updates to methods whose documentation did not change. In the study, we identified 4,769 SEMs, which include 2,271 publicly-accessible methods. After manually analyzing a statistically significant sample containing 562 SEMs, we found that 363 of the methods include semantic changes. Furthermore, we also analyzed the use of SEMs in a sample of 1,000 real-world Android apps and observed that 957 of these apps use at least one SEM. Interestingly, a number of such usages have been manually mitigated by app developers through API version checks[1], indicating that SEMs could indeed introduce compatibility issues in Android apps. Overall, we believe that our results highlight that Android developers do not always thoroughly document semantic changes in the platform API and that these changes can extensively affect real-world Android apps.

In summary, the main contributions of this paper are:

- A characterization of SEMs across the ten most used versions (at the time we started our study) of the Android API. We analyzed method updates across ten versions of the Android API and manually confirmed that 363 methods out of a sample of 562 publicly-accessible methods contain semantic changes. We also characterized the nature of SEMs and found that developers might have accidentally introduced the majority of them as most of them are evolved only once.
- A study of how SEMs impact Android apps. We investigated whether and how Android apps use SEMs in a sample of 1,000 real-world Android apps. We found that SEMs are commonly used by Android apps, and such usages could indeed lead to compatibility issues as app developers have added checks to prevent the execution of SEMs on certain versions of the Android API.
- A tool to identify SEMs and the experimental data containing the findings of our study. To identify and characterize SEMs, we designed a technique and implemented the approach in a prototype tool called ANDROSEA. ANDROSEA identifies methods across two versions of the Android API that have same signature, same

[1] Version checks are recommended by Google to tackle API-induced compatibility issues. A typical check conforms to the following structure: if `SDK_INT` $< n$, call a method of the older API, otherwise, do something else. `SDK_INT` is the short version for `android.os.Build.SDK_INT` and this value, at runtime, provides the version of the API on which a certain app is running. This check ensures that the method of the API, which may introduce compatibility issues in the app when the app is running on newer versions of the API, will only be invoked if the app is running on API versions that are older than the one where the change was introduced.

method comment, but different method body. The tool and the experimental data are publicly available at https://github.com/MobileSE/AndroSea.

The remainder of this paper is organized as follows. Section 3.2 defines relevant terminology. Section 3.3 presents our study methodology. Section 3.4 details the results of the study. We discuss implications for researchers and practitioners in Section 3.5. Section 3.6 outlines related work. Finally, Section 3.7 provides concluding remarks.

3.2 Terminology and Motivation

This sections introduces the relevant terminology we will use in the rest of this paper. Consider two API versions (or levels): *old API* = $[m_1, ..., m_k]$ and *new API* = $[m'_1, ..., m'_l]$. A method m_i from the old API is a *silently-evolved method* (SEM) if there is a method m'_i in the new API that shares the same signature and the same comment with m_i, but has a different method body with respect to m_i. Among SEMs, we define those methods that are publicly accessible (i.e., methods that are declared as public) as *publicly-accessible silently-evolved methods* (PASEMs). In this paper, we focus on both SEMs and PASEMs as both classes might affect the execution of an android app.

Listing 3.1 provides an example of a PASEM, which shows the evolution of the method `getSqlStatementType` between API level 27 and 28. In the example, the two versions of the method have the same signature (line 7) and the same method comment (lines 1-6), but different method body, as Android developers added new statements to the method in the API level 28 (lines starting with +). In both API versions, the method returns the type of the SQL statement provided as input. However, from API level 28, part of the method behavior is changed. In fact, instead of returning `STATEMENT_ABORT` for any SQL rollback statement as defined in API level 27, the method returns a different value (i.e., `STATEMENT_OTHER`) if the SQL statement aims at rollbacking to a savepoint [86].

```
1 /**
2  * Returns one of the following which represent the type of the given SQL
      statement.
3  * ...
4  * @param sql the SQL statement whose type is returned by this method
5  * @return one of the values listed above
6  */
7 public static int getSqlStatementType(String sql) {
8  String prefixSql = sql.substring(0, 3).toUpperCase(Locale.ROOT);
9  else if (prefixSql.equals("ROL")) {
10 + boolean isRollbackToSavepoint = sql.toUpperCase(Locale.ROOT).contains(" TO
      ");
11 + if (isRollbackToSavepoint) {
12 +  ...
13 +  return STATEMENT_OTHER;
14 + }
15   return STATEMENT_ABORT;
16  }
17  return STATEMENT_OTHER;
18 }}
```

LISTING 3.1: Code snippet extracted by comparing the method `getSqlStatementType` between Android API level 27 and 28.

Unfortunately, due to this behavioral change, apps that use the method could exhibit compatibility issues. Specifically, an app that relies on the method's behavior as implemented in API level 27 might encounter a failure when running on newer versions of the API. Because API developers did not suitably document the change, app developers might not be aware of the change and hence have a low chance of avoiding such compatibility issue. In the rest of this paper, we present a systematic study that carefully analyzes this family of changes across multiple versions of the Android API and investigates to what extent these changes might affect Android apps.

3.3 Methodology

Our analysis is based on the Android framework codebase. This codebase is one of the largest repository made available on Github and contains over 440,000 commits

and nearly one thousand release tags. In this study, we focus our analysis on the ten most recent[2] major version releases of the Android framework, as these versions are the ones that are widely used on user devices. (Older releases were less popular and their distribution accounted for less than 3% of the total distribution.) When selecting a revision for analyzing a major version release, we chose the first release tag associated with each version considered. Table 3.1 reports the details of the versions and revisions we considered. Using the versions listed in Table 3.1, we built nine subsequent version pairs (e.g., version 19 and 21 constitute a version pair), and use these pairs to identify SEMs.

To the best of our knowledge, no readily-available tool exists to detect SEMs. For this reason, we implemented a prototype tool called AndroSea, which identifies SEMs across different version of the Android framework. At a high-level, the tool takes as inputs the repository containing the codebase of the Android framework and the list of framework version pairs that AndroSea should compare. Given this information, the tool analyzes the version control history of the repository to compare methods across different versions of the Android API. In this step, the tool categorizes a method as a SEM if the method has the same signature, the same method comment, but different method body. The output of AndroSea is a list of SEMs for each version pair analyzed.

Fig. 3.1 presents a high-level overview of the AndroSea workflow. As the figure highlights, AndroSea uses three modules to identify SEMs. The three modules are the *repository preprocessing module* (RPM), the *method extraction module* (MEM), and the *SEM identification module* (SIM). We now present the three modules in detail.

3.3.1 Repository Prepocessing Module

Each release of the Android framework codebase contains a large variety of files. In fact, the codebase includes the core implementation of the Android API, the source code of various command-line tools (such as the Android Asset Packaging Tool), the source code of unit tests, and other types of files. Because not all of these files are part of the Android API, the repository prepocessing module analyzes the codebase to identify and select the files that relate to the Android API. The repository preprocessing module performs this task by using a whitelist that we manually constructed, and

[2]When we started the study in March 2020.

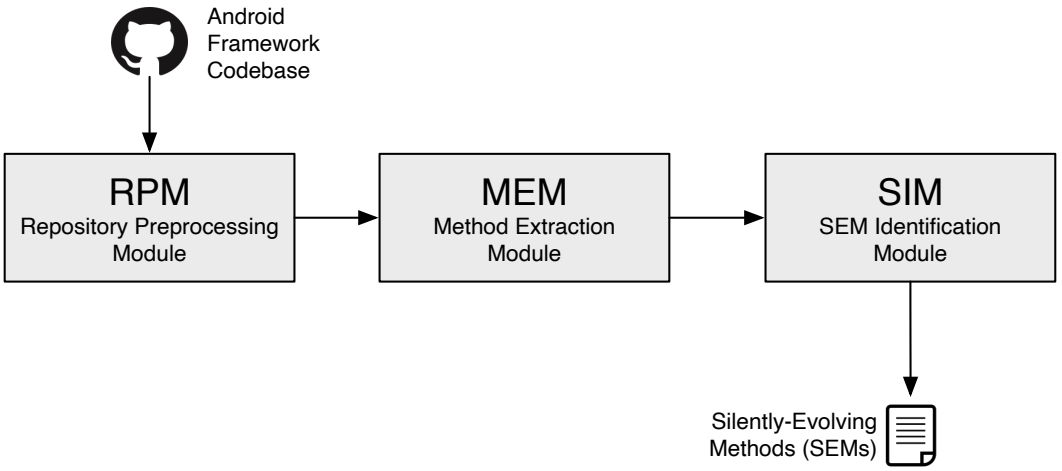


FIGURE 3.1: High-level overview of the ANDROSEA workflow.

TABLE 3.1: Android Framework revisions used in the study. We did not consider API level 20 as this version was focusing on changes for supporting Android Wear.

API Level	Code Name	Release Tag	Distribution
29	Android10	android-10.0.0_r1	8.2%
28	Pie	android-9.0.0_r1	10.4%
27	Oreo	android-8.1.0_r1	15.4%
26	Oreo	android-8.0.0_r1	12.9%
25	Nougat	android-7.1.0_r1	7.8%
24	Nougat	android-7.0.0_r1	11.4%
23	Marshmallow	android-6.0.0_r1	16.9%
22	Lollipop	android-5.1.0_r1	11.5%
21	Lollipop	android-5.0.0_r1	3.0%
19	KitKat	android-4.4_r1	6.9%

performs this step for each framework version provided as input to ANDROSEA. By performing this operation, ANDROSEA reports only the SEMs that are related to the Android API. ANDROSEA provides the list of relevant files to the method extraction module for further analysis.

3.3.2 Method Extraction Module

This module locates and extracts relevant information about the Java methods in the source code files of the Android API. Specifically, for each version of the API, the module creates a set of tuples (*apiInfo*) where each tuple (*mInfo*) represents a method in the API and contains the signature (*signature*), the comment (*comment*), and the body

(*body*) of the method. The module uses a Java parser to build the abstract syntax tree (AST) for each of the source code files and identifies the API methods in a file by navigating the AST. After locating a method, the module stores the method signature, the method comment, and the method body in *apiInfo*. AndroSea only saves the comments declared through the Javadoc notation (i.e., `/**...*/`) since only these comments will appear in the documentation of the API. The output of this module are the sets of tuples that are associated with the versions of the API considered.

3.3.3 SEM Identification Module

The SEM identification module identifies SEMs by comparing the relevant methods in the version pairs provided as input to AndroSea. Algorithm 1 describes how the module identifies SEMs. Given the methods' information from two subsequent versions of the Android API (*apiInfo1* and *apInfo2* in Algorithm 1), the module iterates over the methods in the versions and compares them (lines 1-15). When the algorithm finds methods with matching signatures (i.e., they are the same method in different versions of the API), AndroSea first checks whether the methods have a comment associated with them. If either one of the methods does not have a comment, AndroSea will not consider the methods for further analysis (line 7). The rationale behind this decision is that we believe that such methods might not be intended for use by app developers as they often resort to the official documentation to learn how to use the API methods. If both methods have a comment, AndroSea checks whether the comments are the same or not by comparing their text. If the comments have the same text, AndroSea moves forward and compares their method bodies. For simplicity, AndroSea compares the method bodies using the text of their bodies. If the bodies are different, AndroSea categorizes the method as a SEM and adds the method information to the set of SEMs computed for the API versions pair under analysis.

3.4 Experimental Study

This section discusses our empirical study. In the study, we investigated the following research questions

Algorithm 1: Detecting Silently-Evolved Methods.

Input : *apiInfo1* and *apiInfo2*: method information from two API versions
Output: *sems*: set of silently-evolved methods between the two Android versions considered

```
1  for mInfo1 ∈ apiInfo1 do
2      for mInfo2 ∈ apiInfo2 do
3          if mInfo1.signature ≠ mInfo2.signature then
4              continue
5          end
6          if mInfo1.comment.isEmpty || mInfo2.comment.isEmpty then
7              continue
8          end
9          if mInfo1.comment == mInfo2.comment then
10             if mInfo1.body ≠ mInfo2.body then
11                 sems.add(mInfo1, mInfo2)
12             end
13         end
14     end
15 end
16 return sems
```

- **RQ1:** To what extent do SEMs appear in the Android API?
- **RQ2:** What are the characteristics of SEMs?
- **RQ3:** How do SEMs evolve during the development of the Android API?
- **RQ4:** To what extent are PASEMs used in Android apps?

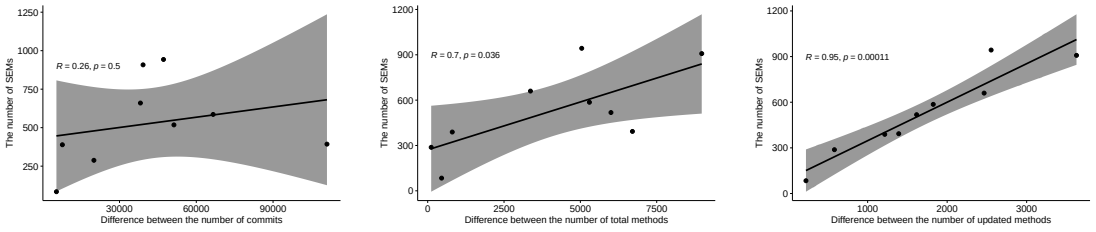
In the rest of this section, we answer the research questions by presenting our experimental findings.

3.4.1 RQ1: SEMs in the Android API

With the first research question, we are interested in quantifying the number of SEMs in the Android API. To this end, we ran AndroSea on the source code of the Android framework codebase using the list of release tags shown in Table 3.1. AndroSea extracted all the methods in each release and conducted a pairwise comparison between each subsequent version pair (e.g., between *android-4.4_r1* and *android-5.0.0_r1*). Since our study considered ten major version releases of the Android API, AndroSea conducted nine pairwise comparisons. In total, AndroSea was able to identify 4,769 SEMs and 2,271 of these SEMs are PASEMs.

TABLE 3.2: SEMs identified in our study categorized by their modifiers and annotations.

API Level	public						default						protected						private					
	-	static	final	abstract	hide	native	-	static	final	abstract	hide	native	-	static	final	abstract	hide	native	-	static	final	abstract	hide	native
19→21	502	92	33	1	142	0	38	6	1	0	6	0	16	0	1	0	24	0	52	5	3	0	39	0
21→22	112	18	4	125	80	0	16	2	1	0	1	0	1	0	1	4	7	0	28	4	1	0	14	0
22→23	279	75	49	0	171	9	15	9	1	0	4	0	7	0	0	0	4	0	63	10	1	0	28	0
23→24	436	80	28	0	336	0	16	9	2	0	5	0	5	0	0	0	4	0	44	13	0	0	15	0
24→25	26	9	5	0	37	0	4	0	0	0	1	0	0	0	0	0	3	0	3	3	0	0	2	0
25→26	233	60	14	0	200	0	19	4	5	0	24	0	3	0	0	0	7	0	32	9	2	0	13	0
26→27	64	25	3	0	193	0	5	2	0	0	5	0	0	0	0	0	0	0	7	2	0	0	2	0
27→28	145	111	15	0	151	0	9	3	0	0	4	0	0	0	0	0	4	0	19	1	0	0	3	0
28→29	166	63	11	0	263	0	4	1	0	0	14	0	2	0	0	0	6	0	29	4	0	0	6	0



(a) (b) (c)

FIGURE 3.2: Correlations between the number of SEMs and the differences of the number of commits, total methods, and updated methods in two subsequent releases, respectively.

After identifying SEMs, we also analyzed the modifiers and annotations associated with the methods to determine the potential impact of the methods on client apps. Table 3.2 reports the number of SEMs identified by ANDROSEA and categorizes them by their modifiers and annotations. The table is divided into five sections. The first section (API Level) reports the information of the version pair considered in the study. The other four sections group methods according to their Java access modifier. In Table 3.2, the columns labeled with the symbol '-' report the number of SEMs that have the Java access modifier as their only modifier. Furthermore, if a method has the Java access modifier and multiple additional modifiers or annotations, we counted the method in all the columns that apply. For example the SEM `getInstance` from the `ConnectivityManager` class in API level 23 is a `private` method that has both the `static` modifier and the `hide` annotation. In this case, we counted the method both in the "static" and the "hide" columns of the "private" section.

The majority of SEMs are declared as public (i.e., they are PASEMs) and these methods can be directly accessed by of Android apps. Based on the update types of PASEMs, these apps may be subject to compatibility issues, which can lead to field failures if the update has changed the method's semantics (e.g., the API method presented in our motivating example and reported in Listing 3.1). We believe that API developers should

pay particular attention to updating these method comments so that app developers can suitably account for the semantic changes affecting their apps.

Although the number of version pairs considered is on the low side (and this might affect the validity of the results), we performed a correlation analysis on the SEMs we identified. Specifically, Fig. 3.2 presents the correlation (obtained via Pearson's correlation coefficient) between the total number of SEMs and the difference between the number of commits (Fig. 3.2 (a)), the number of methods including updated methods (Fig. 3.2 (b)), and the number of updated methods (Fig. 3.2 (c)) in each version pair considered. These correlation results (i.e., Pearson's correlation coefficient R and $p - value$) show that the introduction of SEMs is not strongly correlated with the number of Github commits (which do not necessarily lead to method changes) but strongly correlated with the difference in the number of methods, especially the difference in the number of updated methods, in two subsequent releases.

Answer to RQ1

Based on our empirical results, we can confirm that SEMs are present in the Android API. Considering ten major version releases of the API, we were able to identify 4,769 SEMs, including 2,271 PASEMs. These PASEMs could lead to compatibility issues in their client Android apps. Furthermore, the more methods are updated in a version release, the more SEMs can be introduced in the API.

3.4.2 RQ2: Understanding SEMs

In the second research question, we are interested in understanding the main purposes behind the updates of SEMs. Specifically, we would like to check whether the updates are related to simple code refactorings that would introduce no harm to the system or involved in semantic changes that could break the execution of existing Android apps. Ideally, we would expect that SEMs should only involve code refactorings such as renaming variables and attributes. They should not include semantic changes as those changes could break the execution of client apps. The comments, especially the Javadocs, of the corresponding methods should have subsequently been updated to properly advise app developers to update their apps so as to be aligned with the changed APIs' new semantics.

TABLE 3.3: Manual classification on selected sample PASEMs.

API Level	Sample	Update type				Refactoring	Uncertain	Disagreement
		Added	Removed	Changed	Sum			
19→21	83	16	2	34	52 (62.65%)	22 (26.51%)	9 (10.84%)	22 (26.51%)
21→22	54	17	1	14	32 (59.26%)	19 (35.19%)	3 (5.56%)	1 (1.85%)
22→23	76	23	1	27	51 (67.10%)	21 (27.63%)	4 (5.26%)	21 (27.63%)
23→24	81	29	1	35	65 (80.25%)	13 (16.05%)	3 (3.70%)	9 (11.11%)
24→25	25	8	1	11	20 (80.00%)	2 (8.00%)	3 (12.00%)	5 (20.00%)
25→26	71	15	2	24	41 (57.75%)	23 (32.39%)	7 (9.83%)	11 (15.49%)
26→27	42	13	0	15	28 (66.67%)	14 (33.33%)	0 (0.00%)	2 (4.76%)
27→28	65	9	0	21	30 (46.15%)	24 (36.92%)	11 (16.92%)	12 (18.46%)
28→29	65	13	2	29	44 (67.69%)	10 (15.38%)	11 (16.92%)	14 (21.53%)
Total	562	143	25	199	363 (64.59%)	148 (26.33%)	51 (9.07%)	97 (17.26%)

Unfortunately, to the best of our knowledge, our community has not yet made available tools for effectively determining whether an update of method code (i.e., method diff) is related to semantic change or not. To this end, in this work, we resort to manual efforts to classify the purposes behind the updates of PASEMs (i.e., the nature of the changes). We choose to classify PASEMs instead of SEMs because only PASEMs could directly impact the execution of client apps available in the wild. Since manual efforts are known to be time-intensive, it becomes impractical for us to manually classify all the PASEMs identified previously. To that end, we resort to randomly sample a set of PASEMs to fulfill the purpose. To ensure that the sampled PASEMs are representative, we turn to the well-known online Sample Size Calculator[3] to determine the number of PASEMs for manual classification (with a confidence level of 95% and margin of error of 10%).

```
1 // Between android-8.1.0_r1 and android-9.0.0_r1
2 public CharSequence[] getTextArray(@StyleableRes int index) {
3   final TypedValue value = mValue;
4 - if (getValueAt(index * AssetManager.STYLE_NUM_ENTRIES, value)) {
5 + if (getValueAt(index * STYLE_NUM_ENTRIES, value)) {
```

LISTING 3.2: An example of PASEM flagged as code refactoring.

The second column in Table 3.3 enumerates the number of samples randomly selected from each framework iteration. For each method, two of the authors of the paper independently categorized the method as either *refactoring* or *semantic change*. If the authors cannot make a decision in 10 minutes, the corresponding PASEM will be flagged as uncertain. After completing the independent manual classification, the two authors then set up meetings to discuss their decisions until consensuses reached. The final

[3]https://www.surveysystem.com/sscalc.htm

results are summarized in the last four columns in Table 3.3. In the manual classification, only 97 out of 562 are classified as different types by the two different authors achieving a high inter-rater reliability of 82.74%. After the meeting, 41 out of 97 are concluded as semantic changes accounting for 42.27% while the refactorings and uncertains are made up of 23.71% and 34.02% respectively. Surprisingly, all in all, slightly more than a quarter of the randomly selected PASEMs are related to code refactorings (Listing 3.2 presents such an example), and around two-thirds of the randomly selected PASEMs are related to semantic changes (e.g., logic added, removed, or changed that is complicated update including statements added and removed). This result shows that SEMs are a severe problem in the Android framework. The framework maintainers should pay special attention to carefully handle these methods, i.e., avoid introducing SEMs in future releases and document the semantically changed methods.

```
1 // Between android-9.0.0_r1 and android-10.0.0_r1
2 public void setVolumeTo(int value, int flags) {
3   try {
4 - mSessionBinder.setVolumeTo( mContext.getPackageName(), mCbStub, value, flags);
5 + // Note:  Need both package name and OP package name.  Package name is used for
6 + // RemoteUserInfo, and OP package name is used for AudioService's internal
7 + // AppOpsManager usages.
8 + mSessionBinder.setVolumeTo( mContext.getPackageName(), mContext.getOpPackageName()
      , mCbStub, value, flags);
9   } catch (RemoteException e) {
```

LISTING 3.3: An example of PASEM flagged as uncertain.

As shown in Table 3.3 (the fifth column), around 10% of PASEMs are flagged as uncertain. The majority of those methods are related to updates of callee methods, which may further involve complicated changes. Listing 3.3 presents such an example. The original callee method *setVolumeTo()* called by *mSessionBinder* has been replaced by a new one that involves a new parameter, which is not needed by the original version. The callee method is only defined in a Java interface called *ISessionController*, which is non-trivial for the authors to manually identify its dynamically bound object (in a short time), considering that the Android framework is one of the most complicated open-source projects. Therefore, this update is flagged as *uncertain*.

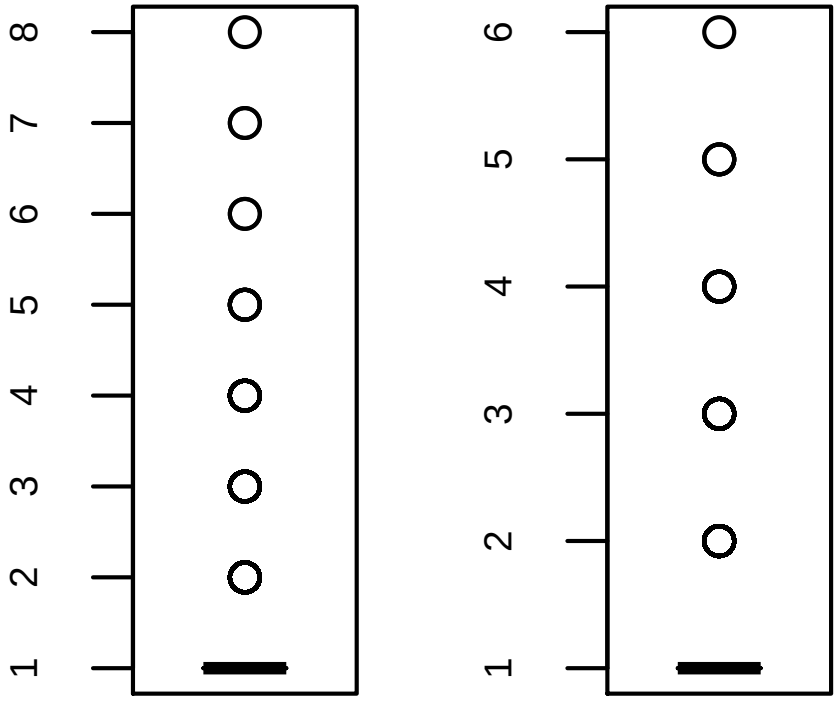
Answer to RQ2

Our manual classification reveals that the majority of PASEMs (over 64.59%) do involve semantic changes, and such changes may involve complicated updates of the code.

3.4.3 RQ3: Evolution of PASEMs

In this research question, we are interested in exploring the evolution of SEMs under the evolution of the Android framework codebase. To this end, we first look at the number of times a given method is silently changed. Fig. 3.3 illustrates the distribution of such times for all the identified 4,769 SEMs and 2,271 PASEMs. Expectedly, the majority of methods (61.12%) that are flagged as SEMs are only silently evolved once, in the meanwhile, 79.03% of PASEMs are silently evolved once, suggesting that SEMs might not be intentionally introduced by the framework developers. Nevertheless, there are several methods that have indeed been repeatedly changed silently. For example, method *loop* of file `core/java/android/os/Looper.java` has been silently updated six times, among the nine considered iterations. The class Looper is used to run a message loop for a thread. For the specific method, it is actually a static public method that is used to run the message queue in this thread.

Among the 4,769 SEMs, interestingly, only 80 of them have their comments updated along with the update of the implementation in the following up revisions of the Android framework. This experimental result suggests that framework maintainers might not yet be made aware of SEMs or at least are not well-motivated to mitigate the introduction of SEMs. Listing 3.4 illustrates one of such updated comments. The implementation of the method has been updated in the release android-5.0.0_r1 while its comments are only updated at the revision android-6.0.0_r1. Nevertheless, although rare, the fact that some of the SEMs are indeed resolved by the framework maintainers shows that SEMs are indeed a problem that should be carefully addressed.



(a) SEMs (b) PASEMs

FIGURE 3.3: Distribution of update times of SEMs from level 19 to 29.

```
1 //android.widget.CheckedTextView.setCheckMarkDrawable
2  /**
3 - * Set the checkmark to a given Drawable. This will be drawn when {@link #
      isChecked()} is true.
4 - * @param d The Drawable to use for the checkmark.
5 + * Set the check mark to the specified drawable.
6 + * <p>
7 + * When this view is checked, the drawable's state set will include
8 + * {@link android.R.attr#state_checked}.
9 + * @param d the drawable to use for the check mark
10 + * @attr ref android.R.styleable#CheckedTextView_checkMark
11   * @see #setCheckMarkDrawable(int)
12   * @see #getCheckMarkDrawable()
13 - * @attr ref android.R.styleable#CheckedTextView_checkMark
14   */
```

LISTING 3.4: The comment of SEM *setCheckMarkDrawable* is updated in a future release while its body is not changed.

We further look at the evolution of method modifiers for all the identified SEMs. Fig. 3.4

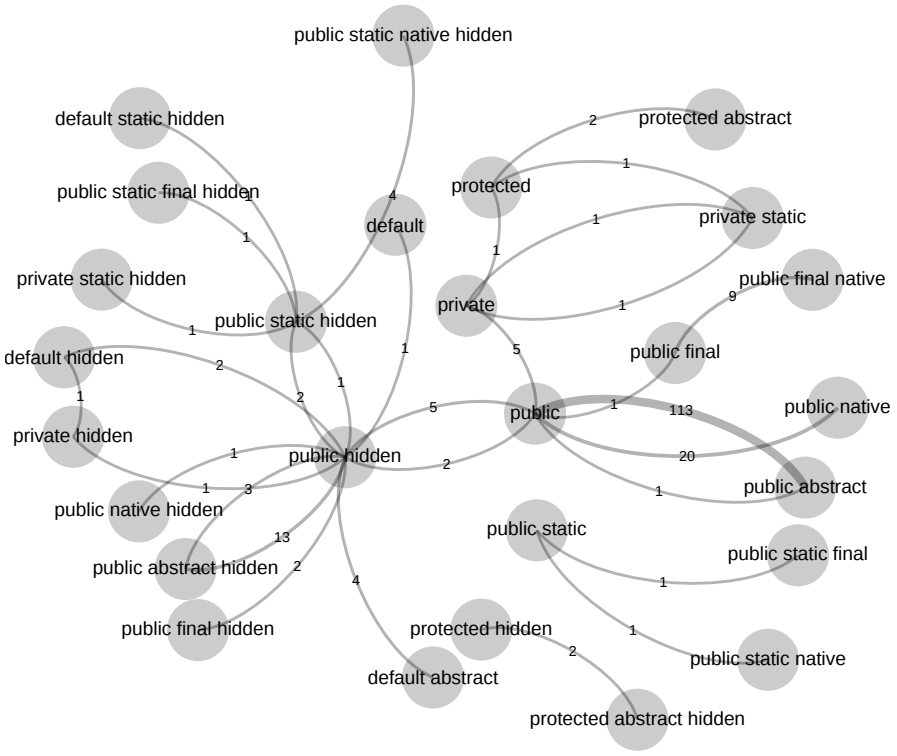


FIGURE 3.4: The update of modifiers. The line in clockwise shows the direction of the update and the weight on the line shows the number of SEMs having such updates.

summarizes such results in a directed graph. Each node represents a distinct modifier type, while each arrowed edge represents an evolution, e.g., from a *src* modifier to the *dest* modifier. The weight of each edge subsequently represents the number of times the corresponding evolution happens during the evolution of the Android framework codebase.

In total, we have observed 204 times of modifier changes. Expectedly, none of the changes are made from *public* to other low-level accessibilities (except one from *public static hidden* to *private static hidden*[4]) as such changes would break the compilation rules for Android apps, resulting in explicit compatibility issues. Interestingly, the majority of modifier changes are related to altering *non-abstract* methods to *abstract* methods or exchanging *native* methods with *non-native* methods. Listing 3.5 represents an example of modifiers update from *public* to *public abstract* extracted between android-5.0.0_r1 and android-5.1.0_r1. The class, the method belongs to, is actually an abstract class. The body of the method is to provide the default behavior, which is to throw `MustOverrideException` if the subclass extends the *WebSettings* but does not

[4]Hidden APIs cannot be directly accessed by Android apps.

explicitly override the method. The update of the method reinforces the need to override the method by declaring it as *abstract*, which utilizes the compile-time check to ensure that the method will be overridden in the subclass. Although this type of change is trivial, it will theoretically cause compatibility issues for existing Android apps. Indeed, previously, even if developers do not override the API method *setBuiltInZoomControls()* when extending the *WebSettings* class, grammatically speaking, there will be no compile error. However, with the latest SDK version, the accessibility of *setBuiltInZoomControls()* has changed to *public abstract*, there will be a compile error if the method is not explicitly overridden. Therefore, ideally, this type of change should be avoided by framework maintainers, and subsequently should be considered by software analyzers aiming at detecting compatibility issues in Android apps.

```
1 // This is an abstract base class:  concrete WebViewProviders must
2 // create a class derived from this, and return an instance of it in the
3 // WebViewProvider.getWebSettingsProvider() method implementation.
4 public abstract class WebSettings {
5 - public void setBuiltInZoomControls(boolean enabled) {
6 -  throw new MustOverrideException();
7 - }
8 + public abstract void setBuiltInZoomControls(boolean enabled);
```

LISTING 3.5: An example of modifier update from public to public abstract.

The majority of the updates related to native methods share the same change pattern, which is to provide a wrapper method with the same name while calling the native method to provide the same behavior (cf. (Listing 3.6). Even though the modifiers were updated from *public native* to *public*, the signatures of the APIs provided to developers are not changed. Practitioners hence can still use the same method signature to implement their intentions. Therefore, this type of update will unlikely to introduce compatibility issues into running Android apps.

```
1 // Code snippet from android-4.4_r1
2 94  public native int getHeight();
3 // Code snippet from android-5.0.0_r1
4 107 public int getHeight() {
5 108  return nativeGetHeight(mNativePicture);
6 109 }
```

LISTING 3.6: An example of modifier update from public native to public.

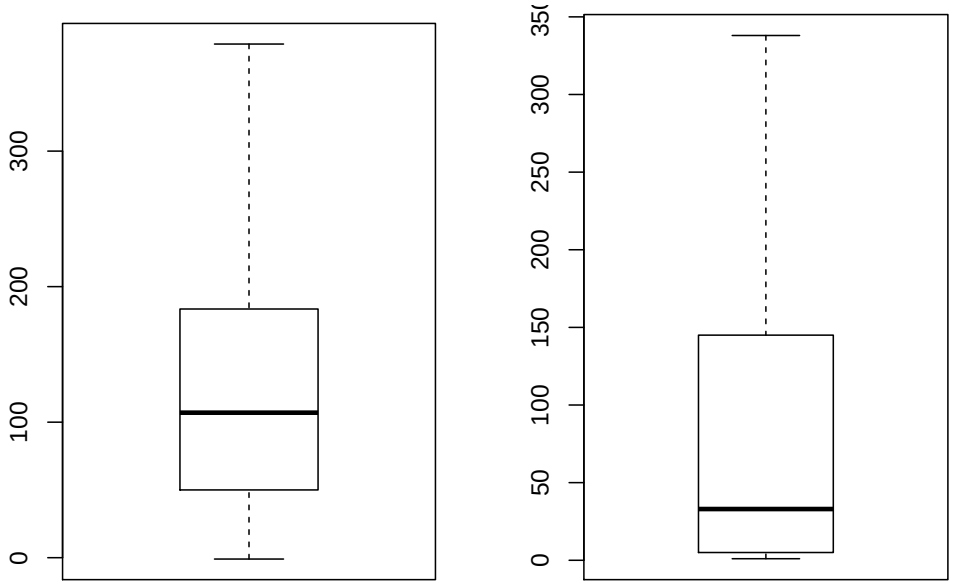
Answer to RQ3

The majority of SEMs is only introduced into the framework once, without following up updates. Moreover, SEMs may involve updating the method's modifiers that could further introduce runtime issues to client Android apps.

3.4.4 RQ4: PASEMs in Android apps

Since PASEMs may involve semantic changes, their client apps may suffer from incompatible issues (because of those silent semantic changes) when they are running on devices with different SDK versions. Therefore, in this last research question, we are interested in knowing if PASEMs are used by Android apps. If so, to what extent are they accessed, and what are the potential impacts such usages could bring to Android app developers? To answer these questions, we introduce a simple app scanner to the community, which takes as input an Android APK and the list of PASEMs identified in this work and outputs the list of PASEMs that are actually accessed by the APK. The app scanner is implemented on top of the famous Soot analysis framework [87]. The APIs are identified at the Jimple level, which is one of the intermediate representation types provided by Soot to ease the analysis of Android apps.

Specifically, to fulfill our experiments, we randomly select 1,000 apps published in 2020 from the official Google Play store. All of the selected apps are then sent to the aforementioned app scanner to check whether they have leveraged PASEMs or not. Interestingly and surprisingly, among the 1,000 apps, 957 of them have accessed PASEMs. Moreover, in total, 44.25% PASEMs are accessed. Fig. 3.5(a) further presents the distribution of the number of PASEMs accessed by these apps, giving a median and average number of PASEMs at 107 and 126, respectively. This experimental result shows that PASEMs have been significantly accessed by Android apps, which could subsequently suffer from "hidden" incompatible issues. This result strongly suggests that the Android framework maintainers should pay special attention to avoid introducing PASEMs. This finding is further backed up by the fact that semantically updated PASEMs are also significantly accessed by Android apps, as illustrated in Fig. 3.5(b), for which only the 363 manually confirmed PASEMs (involving semantic changes) are considered.



(a) Total PASEMs. (b) Semantically updated PASEMs.

FIGURE 3.5: The distribution of the number of PASEMs used in a set of randomly selected 1,000 APKs published on 2020.

Previous studies, such as the one proposed by He et al. [10], shows that the convention to address the evolution-induced API compatibility issue is to add additional conditions to check the practical running API level of the device (e.g., through the default constant value of `VERSION.SDK_INT`) before invoking the corresponding APIs. In this work, we try to detect how many different PASEMs are called after the API level condition check, where the PASEMs are referred to as protected PASEMs. Interestingly, 469 out of 1,000 Apps indeed contain protected PASEMs, for which there are 134 PASEMs called under protection. Table 3.4 presents the top-10 PASEMs ranked by their protection times. The fact that the top-ranked PASEMs are protected more than 800 times shows that PASEMs could be protectively accessed many times within the same app.

Moreover, we go one step deeper to check the actual API levels leveraged for protecting the invocation of PASEMs. Our experimental results reveal that a large portion of PASEMs (i.e., 45 or 33.58%) are protected by different API levels, indicating possible errors of app developers, although they have attempted to protect the accessed PASEMs. The possible reasons behind these disagreements between developers could be that PASEMs are silently (hence hiddenly) introduced to the framework. There is hence no documentation for developers to correctly use these PASEMs. Subsequently, developers have to independently identify the API levels suitable for protecting PASEMs

TABLE 3.4: The top ten protected PASEMs in 1,000 APKs

PASEM	times
android.content.res.TypedArray: void recycle()	848
android.os.Bundle: void putParcelable(String,Parcelable)	788
android.os.Bundle: void putCharSequence(String,CharSequence)	623
android.text.TextUtils: void writeToParcel(CharSequence,Parcel,int)	384
android.app.AlarmManager: void setExact(int,long,PendingIntent)	359
android.app.AlarmManager: void setExactAndAllowWhileIdle(int,long,PendingIntent)	289
android.app.Activity: boolean navigateUpTo(Intent)	286
android.app.Activity: void startIntentSenderForResult(IntentSender,int,Intent,int,int,int,Bundle)	285
android.view.ViewGroup: void removeView(View)	283
android.os.Bundle: void putAll(Bundle)	273

independently, based on their empirical evidence. The fact that developers have attempted to protect the access of PASEMs (although may incorrectly do so because of lacking documentation) indicates that PASEMs can indeed introduce runtime issues to Android apps. Therefore, we argue that PASEMs should be totally avoided by framework maintainers. There is also a need to introduce automated tools to regulate that. Our approach ANDROSEA could be leveraged to achieve such a purpose.

Answer to RQ4

PASEMs have indeed commonly been accessed by real-world Android apps. Some of them are even accessed with protections (by checking the running API level), indicating that practitioners have realized that those PASEMs could introduce runtime issues to their apps, although they are not documenting.

3.5 Discussion

We now discuss the potential implications of this study for both practitioners and our fellow researchers, as well as some promising future research directions that could be built on the findings of our research (cf. 3.5.1). After that, we present the potential threats to the validity of this study (cf. Section 3.5.2).

3.5.1 Implication for Practitioners and Researchers

As shown in Figure 3.5, the usages of PASEMs are common in real-world Android Apps, and over 60% of the PASEMs involve truly semantic changes, as demonstrated in Table 3.3. As semantic changes could introduce potential crashes (or security, efficiency issues) in daily use of the Android Apps, we argue that framework maintainers

should try their best to avoid introducing PASEMs. This should also apply to the maintenance of any other third-party frameworks or libraries that provide APIs to facilitate the development of client apps. Subsequently, the client app developers should pay special attention to those silently evolved methods when developing their apps.

As shown in Table 3.3, slightly more than a quarter of SEMs do not involve semantic changes but are simply related to code refactorings. When performing compatibility analyses, these methods could be ignored as they will not introduce runtime issues to their users (i.e., client apps). However, to the best of our knowledge, our community has not introduced promising tools to automatically decide if a given code diff involves semantic changes. State-of-the-art refactoring detection tools [88, 89] are not capable of accurately achieving that. Therefore, we argue that there is a strong need to invent an automated approach to locate semantic changes during the evolution of software systems. With the help of this tool, SEMs with semantic changes could be automatically identified and thereby mitigated by codebase maintainers.

Besides the implementation update of the methods, the methods' modifiers might also be updated along with. The update of the modifiers also could bring in big problems, such as the example in listing 3.5. Therefore, we argue that our fellow practitioners should pay attention to the updates of methods' modifiers when updating their software systems.

3.5.2 Threats to Validity

As it is the case of most empirical evaluations, there are threats to validity associated with the results we presented. In terms of external validity, our results might not generalize to other version pairs in the Android framework. In particular, we identified SEMs in only nine versions pairs. This limitation is an artifact of the complexity associated with the manual analysis performed in the study. To mitigate this threat, we considered all the widely-used major version releases of the Android framework. In relation to external threats to validity, the results presented for answering RQ4 might not generalize to other apps. We mitigated this threat by randomly selecting apps from AndroZoo, one of the most comprehensive app datasets made available to the research community. Finally, an additional threat to validity could be posed by the fact that our study involves manual tasks, which could have introduced errors in the results we presented. For example, in

our study, we had to manually inspect and summarize the code associated with SEMs. To mitigate this threat, two authors cross-validated and inspected the results.

3.6 Related Work

3.6.1 API evolution

While API evolves to meet new feature requirements, to fix bugs etc., developers need to update their implementation of the Application and publish the newer version to provide a stable running environment for their customers. McDonnell et al. [12] conducted an extensive empirical study on the stability and adoption of Android APIs while focusing on the relationship between the API evolution and client adoption. The authors in the paper confirm that the Android API evolves more frequently than the client adoption and what's more, the more frequent update of the APIs the longer time for client to adopt the ones. Bavota et al. [13] and Linares-Vásquez [25] studied the relationship between the popularity of the Android Applications and the stability of the SDK APIs. Their empirical study reveal that the more enjoyable Android Apps are prone to call the less updated APIs. Works [14] and [10] investigate the API-related compatibility issues. Li et al. [14] present an approach CiD to highlight the API usage that can lead to potential compatibility issues by analysing the framework release history and identifying the methods without API level checking. He et al. [10] investigate the evolution-induced compatibility issues in Android Applications. Their research shows that the Android Support library only provides limited support for the new APIs in each release and the majority of the Applications need to handle the evolution-induced compatibility issues in their own implementation. Different from the existing work focusing on the general APIs, what we do is to disclose the silently evolved methods that always ignored by developers and researchers.

3.6.2 API pattern

In addition, researchers propose many different approaches to detect Android malware to address the security problems of remote control, privilege escalation, and privacy leakage etc. Chan et al. [28] proposed a static approach for Android malware detection

via extracting permissions and API usage. They confirm that the integration between the feature of permission and API calls can achieve a better precision than just the only feature of permission. Karbab et al. [90] introduced an automatic and effective Android malware detection system, MalDozer, that depends on deep learning techniques and raw sequences of API method calls. Arp et al. [91] proposed a tool DREBIN that builds a SVM based detection model utilizing APIs and other related information. While Ma et al. [92] de-compile the Android Apps and construct three different system API data sets: API usage, API frequency, and API sequence to detect malware. To be specific, Linares-Vásquez et al. [27] attempted to reveal the APIs and usage patterns related to energy consumption, and to provide potential guidance for developers to decrease energy consumption.

3.6.3 Special APIs

Android APIs typically follow the general *deprecated-replace-remove* evolution cycle. Work [15] introduced prototype tool called CDA to characterize deprecated APIs from different revisions. Their extensive investigation shows that the deprecated APIs are not continuously annotated and documented and over a half of these APIs are commented to provide alternatives but these alternatives are rarely replaced by the developers. Besides the aforementioned general publicly accessible APIs, there exists another type of API referred to as inaccessible API that can be recognized as internal or hidden. Internal APIs are resolved ones for system apps located in the package `com.android.internal` while hidden APIs are methods annotated by the javadoc `@hide`. Li et al. [16] did an extensive investigation to reveal the usability of these APIs. They demonstrate that these inaccessible APIs are continuously implemented in the Android framework and used to access a specific set of features while without any promise of forward compatibility. They also reveal that there exist a plenty of apps are indeed calling these inaccessible APIs and the patterns of usage are quite different between each other.

3.7 Conclusion

In this paper, we presented an empirical study that investigates silently-evolved methods (or SEMs in short) in nine version pairs of the Android API. To perform the study, we built a prototype tool called AndroSea. This tool, given the Android framework codebase as an input, is able to identify SEMs in the Android API. Using AndroSea, we found that SEMs are indeed present in the Android API, and a large number of them can be publicly accessed by app developers. Additionally, we empirically found that (i) the majority of publicly-accessible silently-evolved methods (or PASEMs in short) involve semantic changes and these changes could lead to field failures in their client apps, (ii) the majority of SEMs is only introduced into the framework once, indicating that framework maintainers may not have been aware of this situation, and (iii) PASEMs are frequently used by real-world Android apps, even without suitably using API version checks.

We foresee a number of venues for future work. First, we plan to run a user study in which we investigate how developers react to changes in the Android API and their related documentation. We believe that such a study would provide a better understanding on how to best document semantic changes in the Android API. Second, we plan to study how SEMs are related to bug reports by mining and analyzing issues on GitHub. Finally, we plan to investigate whether Android developers follow different development practices in minor and major version updates of the Android framework and how these practices relate to SEMs.

Chapter 4

Co-evolution of Android OS Customizations and Official Releases

Pei Liu, Mattia Fazzini, John Grundy, and Li Li. ”Do customized Android frameworks keep pace with Android?.” In Proceedings of the 19th International Conference on Mining Software Repositories, pp. 376-387. 2022.

4.1 Introduction

Today, we use mobile devices for many of our daily activities, such as reading the news, doing online shopping, streaming content, and communicating with family and friends. One common trait of mobile devices is that they rely heavily on their operating system (OS). In fact, mobile devices use the OS to manage applications (apps), facilitate inter-app communication, and allow apps to access a device's hardware. The Android OS is currently the most widely used OS across mobile devices, having 87% of the market share [93] and running on more than two billion devices [94]. One reason behind Android's popularity is that the OS is an open platform. This characteristic enables device vendors and OS providers to satisfy a large variety of customer needs with customized OS versions.

To create a customized version of the Android OS, device vendors and OS providers modify their own copy of the source code of the Android OS, whose official version is available in the repository managed by the Android Open Source Project (AOSP) [95]. When device vendors and OS providers create customized versions of the Android OS, they usually add new features but may also modify or even delete existing ones. These customizations often contain changes that significantly differ from the original code of the Android OS, leading to divergent versions of the Android OS. Even though these customizations are divergent versions of the Android OS, device vendors and OS providers periodically need to update their customizations so that they can integrate the changes from the Android OS, which provide bug fixes, security patches, and new functionalities [15, 16, 18]. To this end, developers from customized OSs perform merge operations based on the new releases of the Android OS. Because such merge operations can span across the changes characterizing the customized OSs, these operations can, unfortunately, lead to merge conflicts.

While software developers have always experienced some form of merge conflicts in non-divergent team projects, the conflicts experienced in the customizations of the Android OS might be particularly challenging to be resolved. This situation can emerge because (i) the changes in the Android OS happen without any knowledge of the developers working on the customized OSs; (ii) the Android OS evolves at a very rapid pace [12, 13, 25, 81–83]; and (iii) new releases of the Android OS frequently contain thousands of commits. Although related work has analyzed how some of the changes in one customization of the Android OS compare to some of the changes in the Android OS [51], there is still little understanding on whether customized OSs merge the changes from new releases of the Android OS and the characteristics of the actual merge operations. Furthermore, we also do not yet know how different customizations of the Android OS perform this type of merge operation and whether merge conflicts are a recurring problem in different customizations.

To bridge this gap, we present an empirical study that analyzes the version control history of eight open-source customizations of the Android OS and investigates how their developers merged the changes from the new releases of the Android OS into their projects. The study focuses on the portion of the Android OS that provides the Android framework base (as this part contains key OS services that apps directly and heavily

rely on [12, 13, 25, 81–83, 96]). The study investigates how frequently developers performed this type of merge operations and the properties of the operations. Furthermore, to provide a perspective on the potential effects of these conflicts, we also analyzed a randomly selected sample of 1,000 apps and studied how many apps use methods that are affected by conflicts. Our results show that (1) developers performed this type of merge operation for 9.7% of versions released by the Android OS, (2) developers have encountered at least one conflict in 41.3% of the merge operations performed, (3) source code methods are the code entities most affected by the conflicts, (4) 58.1% of the conflicts required developers to change the customized OSs, and (5) 64.4% of the considered apps use at least one method affected by a conflict.

The large number of conflicts identified and the low percentage of merge operations performed motivate further research aiming to help developers perform these operations. Furthermore, the high number of conflicts and the high percentage of apps using methods affected by conflicts indicate that these apps might also experience a range of compatibility issues (i.e., issues that prevent apps from working as expected when running on customized OSs [10, 14, 19–22]) and further research is needed to help app developers handle these issues. The paper elaborates on our findings to help researchers and practitioners who work with the Android OS and guide future research on the topic.

This paper presents an empirical study of how different customizations of the Android OS merge changes to account for new releases of the Android OS. The paper also analyzes the extent to which the conflict-affected methods are accessed in Android apps. Our findings and their implications can help in the design of automated or semi-automated techniques for better supporting developers of customized Android OSs. Finally, to support future research, we make our study infrastructure and results publicly available in our online appendix [97].

```
1        ...
2        if (mConfig.show4gForLte) {
3         mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.FOUR_G);
4        } else {
5         mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.LTE);
6        }
7        mNetTIL.put(TM.NETWORK_TYPE_IWLAN, TI.WFC);
```

```
8       ...
```

LISTING 4.1: Part of the `mapIconSets` method in the Android OS before the update.

```
1       ...
2        if (mConfig.show4gForLte) {
3          mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.FOUR_G);
4       +  if (mConfig.hideLtePlus) {
5       +   mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G);
6       +  } else {
7       +   mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G_PLUS);
8       +  }
9          } else {
10         mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.LTE);
11      +  if (mConfig.hideLtePlus) {
12      +   mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.LTE);
13      +  } else {
14      +   mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.LTE_PLUS);
15      +  }
16         }
17         mNetTIL.put(TM.NETWORK_TYPE_IWLAN, TI.WFC);
18      ...
```

LISTING 4.2: Changes to `mapIconSets` in the Android OS after the update.

```
1       ...
2       if (mConfig.show4gForLte) {
3       <<<<<<< HEAD
4        if (mContext.getResources()
5         .getBoolean(R.bool.show_4glte_icon_for_lte)) {
6         mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.FOUR_G_LTE);
7        } else if (mContext.getResources()
8         .getBoolean(R.bool.show_network_indicators)) {
9         mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.LTE);
10       } else {
11        mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.FOUR_G);
12       }
13       mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G_PLUS);
14      } else {
15       mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.LTE);
```

```
16          if (mContext.getResources()
17           .getBoolean(R.bool.show_network_indicators)){
18           mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G_PLUS);
19          } else {
20           mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.LTE);
21         =======
22          mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.FOUR_G);
23          if (mConfig.hideLtePlus) {
24           mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G);
25          } else {
26           mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G_PLUS);
27          }
28         } else {
29          mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.LTE);
30          if (mConfig.hideLtePlus) {
31           mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.LTE);
32          } else {
33           mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.LTE_PLUS);
34         >>>>>>> android/nougat-mr1.6-release
35          }
36         }
37         ...
```

LISTING 4.3: Merge conflict affecting `mapIconSets` in the LINEAGEOS.

```
1            ...
2            if (mConfig.show4gForLte) {
3             if (mContext.getResources()
4              .getBoolean(R.bool.show_4glte_icon_for_lte)) {
5             mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.FOUR_G_LTE);
6             } else if (mContext.getResources()
7              .getBoolean(R.bool.show_network_indicators)) {
8             mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.LTE);
9             } else {
10            mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.FOUR_G);
11            if (mConfig.hideLtePlus) {
12             mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G);
13            } else {
14             mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G_PLUS);
```

```
15              }
16             }
17             mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G_PLUS);
18            } else {
19             mNetTIL.put(TM.NETWORK_TYPE_LTE, TI.LTE);
20             if (mContext.getResources()
21              .getBoolean(R.bool.show_network_indicators)){
22              mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.FOUR_G_PLUS);
23             } else {
24              if (mConfig.hideLtePlus) {
25               mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.LTE);
26              } else {
27               mNetTIL.put(TM.NETWORK_TYPE_LTE_CA, TI.LTE_PLUS);
28              }
29             }
30            }
31            ...
```

LISTING 4.4: Conflict resolution in the LINEAGEOS.

4.2 Terminology & Motivation

This section introduces some relevant terminology and presents an example that we use to motivate our work.

4.2.1 Terminology

Given a customized version of an OS, we call the project of the customized OS the *downstream project* and the project of the original OS as the *upstream project*. The downstream and the upstream projects are maintained in two different repositories, and we use the terms *downstream repository* and *upstream repository* to refer to these repositories. The downstream and the upstream projects use a version control system (VCS) to manage the changes associated with the files contained in the repositories. *Downstream developers* make changes to the files in the downstream repository, while *upstream developers* edit files in the upstream repository. Periodically, downstream

developers pull changes from the upstream repository and merge them into the downstream repository. Downstream developers perform the *merge operations* using the VCS. While performing merge operations, downstream developers can experience *textual merge conflicts*. In this work, we use *merge conflict* as an abbreviation for textual merge conflicts. A merge conflict can appear when the VCS cannot create a merged file given the changes to the file in the downstream and upstream repositories. Downstream developers resolve conflicts by suitably editing the conflicting changes associated with the file of the downstream repository.

4.2.2 Motivation

Listings 4.1 - 4.4 provide an example[1] of a merge conflict appearing in the LINEAGEOS [98], an open-source customization of the Android OS that offers custom OS management and security features [99, 100]. In this example, the LINEAGEOS is the downstream project and the Android OS is the upstream project. The conflict affects the method `mapIconSets` in the `MobileSignalController.java` file, which produces a mapping of data network types to icon groups. The conflict appeared when the downstream repository merged the changes from the upstream repository that have the `nougat-mr1.6-release` tag, which represents the changes characterizing a new version of the upstream repository. The conflict appears because the method `mapIconSets` in the downstream repository (lines 4-20 in Listing 4.3) has a different implementation with respect to the same method in the upstream repository (lines 22-33 in Listing 4.3).

Downstream developers needed to handle such a conflict with careful attention because the part of the conflict from the upstream repository contains an update. Listing 4.1 and 4.2 provide the details of the update. Specifically, the method `mapIconSets` was updated by the upstream developers to handle new cases in the mapping of data network types to icon groups (lines 4-8 and lines 11-15 in Listing 4.2). Downstream developers identified these changes and suitably updated their implementation of the `mapIconSets` method as reported in Listing 4.4 (lines 11-15 and lines 24-28).

This example presented how developers from LINEAGEOS updated their downstream project to account for a change in the Android OS. In the rest of this paper, we present

[1] We shorten variable names in the example due to space limitations.

our study that characterizes whether and how developers from customizations of the Android OS update their downstream projects to account for the changes in the corresponding upstream projects.

4.3 Study Design

In our study, we first aim to understand how often and how promptly developers from customized OSs merge changes from upstream repositories by answering the following research questions (RQs):

RQ1: Do downstream developers merge changes from corresponding upstream projects?

RQ2: What is the time lag between merge operations and corresponding commits in upstream repositories?

When merging changes from upstream projects, the merge operations might lead to conflicts that need to be explicitly resolved by the developers of the downstream projects. We experimentally understand how often such conflicts happen and how they are resolved by answering the following RQs:

RQ3: Do downstream developers experience conflicts when merging changes from upstream projects?

RQ4: What types of code entities are affected by the merge conflicts?

RQ5: What is the size of method-related conflicts?

RQ6: How often do downstream developers change files in their projects when they experience a merge conflict?

Merge conflicts might indicate that customized projects might have diverged from their corresponding upstream project. Such differences may introduce compatibility issues into Android apps, which are developed based on a single framework version (often the official Android framework). We investigate the potential impact of merge conflicts with respect to compatibility issues by answering the following RQ:

RQ7: Do developers of Android apps use methods affected by merge conflicts?

4.3.1 Dataset Selection

To identify relevant downstream projects, we analyzed a readily available and curated list of customized OSs based on Android [101]. To the best of our knowledge, the list is an up-to-date list of custom OSs. The page is still under active maintenance and the latest update on the page was on March, 2022. At the time of writing, the list contains 33 projects. When we processed the list, we looked for projects that (i) customized the Android framework base, (ii) offered public access to the source code of their customization, and (iii) had their source code stored in repository using a version control system. Our study focuses on the Android framework base for two reasons. First, the Android framework base contains the implementation of core services in the OS. Second, we could readily investigate how the changes in this part of the OS affect Android apps, as apps heavily rely on this part of the framework. These selection criteria left us with nine candidate projects. Of the 24 projects we excluded from consideration, three projects were discontinued, and 21 projects did not provide access to the source code of the Android framework base. Among the remaining nine projects, we observed that one project (called /E/ [102]) only performed one update to merge changes from the Android OS and decided to exclude the project from the study as it would have not significantly contributed to the results of our study.

Table 4.1 lists the remaining eight projects, which are the benchmarks we used in our study. The table reports the name of downstream project (column *Downstream PN*), the name of the corresponding upstream project (column *Upstream PN*), the link to the source code of the downstream project (column *Repository URL*), and the number of stars on Github (column *#. Stars*). All the downstream projects except CRDROIDANDROID have the Android OS as their upstream. The upstream project for CRDROIDANDROID is LINEAGEOS and we decided to include this downstream project to also investigate the properties characterizing customized OSs deriving from customizations of the Android OS. To the best of our knowledge, these projects have been quite popular as they all have a significant number of stars on Github. (Replicant is not hosted on Github and, therefore, we could not collect the number of stars for the project.) Additionally, AOKP was used on more than 3.5 million devices in 2013 [103].

TABLE 4.1: Downstream projects considered in the study.

Downstream PN	*Upstream PN*	*Repository URL*	*Stars*
AOKP	Android OS	https://git.io/J3zId	31
AOSPA	Android OS	https://git.io/JnXSE	96
CRDROIDANDROID	LineageOS	https://git.io/J3zLk	64
LINEAGEOS	Android OS	https://git.io/J3zL0	331
OMNIROM	Android OS	https://git.io/J3zLn	105
REPLICANT	Android OS	https://bit.ly/3gZkdoE	-
RESURRECTIONREMIX	Android OS	https://git.io/J3zLS	111
SLIMROMS	Android OS	https://git.io/J3zL5	40

4.4 Study Results

We answer the RQs of the study. For each RQ, we first describe the methodology used to answer the RQ and then present the results.

RQ1: *Do downstream developers merge changes from corresponding upstream projects?*

Methodology: To answer RQ1, we analyzed the version control history of the downstream projects and identified commits that merged updates from the corresponding upstream repositories. To automatically identify these commits, we first manually inspected the version control history of the repositories and identified that these commits have two parent commits (which was expected as the commits are merge commits) and the commit *identifier* (also known as the commit *hash*) of one of the two parents was also present in the corresponding upstream project. Moreover, some of the merge commits also contain commit messages describing the merge operation from the upstream, which we used to validate our manual inspection. Listing 4.5 provides an example of a relevant merge commit from the LINEAGEOS.

```
commit 80d8b6467ec454e76eb69eb49f80f74d1e
Merge: 9b48a29cca52 37a24f52e6be
Author: xxx<xxx@xxx.com>
Date:   Sat Nov 16 08:46:50 2019 -0700
Merge android-10.0.0_r11 into lineage-17.0
Android 10.0.0 release 11
Change-Id: I05fb998d2e35db534880c89921f595d2225dc9a2
```

LISTING 4.5: Commit merging changes from the Android OS.

TABLE 4.2: Merge operations in downstream projects.

Downstream Project Name	*Total Downstream Commits*	*Downstream Specific Commits*	*Merge Commits*	*Tags*	*Tags (%)*
AOKP	374,786	4,716	15	14	2.50%
AOSPA	512,846	12,370	256	53	9.46%
CRDROIDANDROID	517,912	8,517	351	-	-
LINEAGEOS	521,435	20,539	163	136	24.29%
OMNIROM	498,703	5,707	251	92	16.43%
REPLICANT	213,805	6,937	34	22	3.93%
RESURRECTIONREMIX	441,335	14,585	45	37	6.61%
SLIMROMS	429,688	8,081	33	28	5.00%
Total	3,510,510	81,452	1,148	382	-

The listing reports the identifier for the merge commit (`80d8b6467 ec4` at line 1) and the identifiers associated with the parent commits (line 2). The first commit identifier (`9b48a29cca52`) across the two parents denotes a commit in the LINEAGEOS, while the second commit identifier (`37a24f52e6be`) corresponds to the commit containing the changes that the downstream repository merged from the upstream repository. Given these characteristics, in our automated analysis, we identified relevant merge commits by checking whether a merge commit had one of the parent commits whose identifier also appeared in the corresponding upstream repository. After identifying a relevant merge commit, we also checked whether the parent commit from the upstream repository had a tag associated with the commit. We identified this information by retrieving the list of tags and corresponding commit identifiers from the upstream repository. We retrieve this information as tags identifying version releases in the Android OS. For example, the tag `android-10.0.0_r1` in line 5 of Listing 4.5 represents a version of the Android system. For CRDROIDANDROID we did not retrieve this information as its upstream repository (LINEAGEOS) stopped using tags in 2015 [104].

Results: Table 4.2 reports the results of running our analysis on the eight downstream projects. For each of the downstream projects, the table provides the total number of commits in the downstream project (column *Total Downstream Commits*), the number of commits specific to the downstream project (column *Downstream Specific Commits*), the number of commits that merge updates from the upstream projects (column *Merge Commits*), the number of commits having a tag (column *Tags*), and the percentage of

tags as compared to the total number of tags[2] identifying a version release in the corresponding upstream project (column *Tags (%)*). The difference between the total number of downstream commits and the number of downstream-specific commits is due to the fact that merge operations bring a large number of commits from upstream repositories into downstream repositories. The number of downstream-specific commits also shows that the projects are characterized by substantial development.

The results reported in Table 4.2 show that **there is a high variance in the number of merge operations across the downstream projects**. For example, developers from LINEAGEOS performed 163 merge operations, while developers from AOKP performed this type of operations only 15 times, even if the projects started at similar times (2009 for LINEAGEOS and 2011 for AOKP) and are still active. The percentage of tags associated with the merge operations also varies significantly. For example, in AOKP, 93.3% (14/15) of the merge commits have a tag associated with them, while in AOSPA, only 20.7% (53/256) of the commits are associated with a tag. Overall, the downstream projects performed 1,148 merge commits to merge updates from their upstream repositories, but these commits do not always have a tag associated with them. In fact, only 47.9% of the commits are associated with a tag, meaning that in roughly half of the cases, the commits are not associated with a new version release of the upstream project but some intermediate version of the upstream project. We believe that this result paves the way for new research on automatically identifying and suggesting to developers which updates to merge in their downstream projects. We believe that automated techniques based on static analysis and machine learning could compare downstream and upstream changes to identify the proper time to merge updates from upstream repositories.

The results in Table 4.2 also show that **downstream projects merge updates only for a small percentage (9.7%) of the versions released by the upstream projects**. Fig. 4.1 provides a plot of the merge operations over time. The plot reports the number of merge commits performed by each of the downstream projects in the years from 2009 (the year the first downstream project was created) to 2020 (the year we started this study). All projects have merged updates from their upstream projects for at least three consecutive years. Furthermore, five projects have never stopped merging updates since they were created. The plot also shows that AOSPA, OMNIROM, and

[2]We identified the tags from the Android OS documentation [105].

LINEAGEOS are the projects with the highest number of merge operations in the recent years.

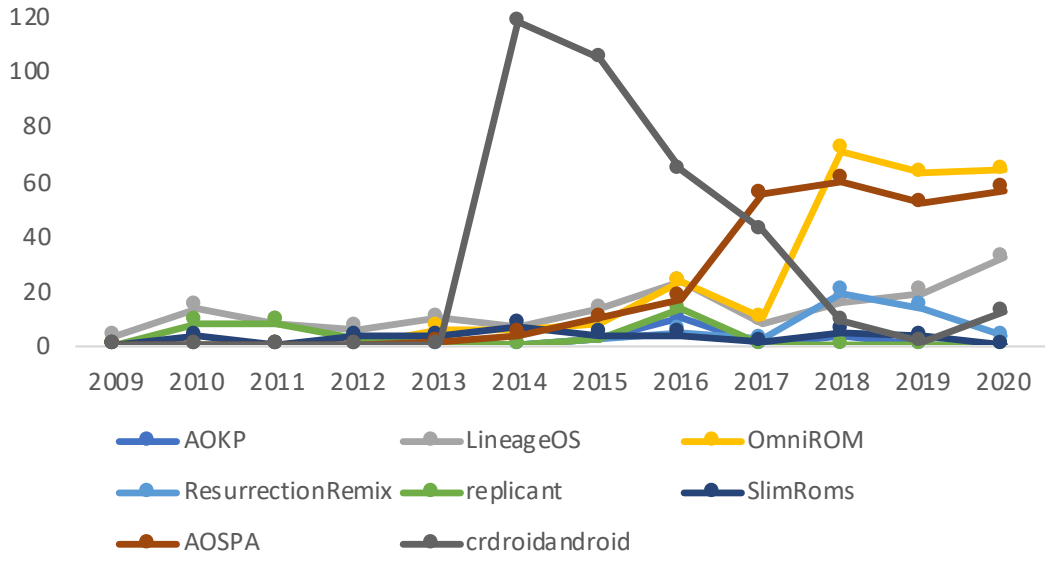


FIGURE 4.1: Distribution of merge commits over time.

To further detail how developers have merged updates over time, Fig. 4.2 plots the number of merge commits that have an associated tag over time. The plot reveals that **downstream projects do not consistently merge updates from new releases of the upstream projects**. However, the LINEAGEOS is increasing the number of this type of merge commits in recent years. Additionally, after a manual inspection of the tags, we did not identify any patterns in the tags that are merged by any of the projects. For instance, the projects did not always merge the first or the last release tag associated with a major version release of the corresponding upstream project. Android has 18 major releases to date [105]. We expected to see these tags used frequently as they indicate a new major revision of the official upstream repository or the latest update, which could provide long term support. Overall, the plots seem to indicate that, currently, downstream projects do not use a systematic approach (periodically merge or merge as soon as a new major version is released) to determine when to merge updates from upstream repositories.

RQ1 Findings

Downstream projects merge updates from upstream projects. The merge operations are both based on commits identifying version releases in the upstream projects (47.9% of the case) but also on other commits (52.1% of the cases). Additionally, downstream projects perform merge operations only for a small portion (9.9%) of all the version releases in the upstream projects. Finally, the results seem to indicate that downstream projects do not use a systematic approach to decide which updates to merge from upstream repositories.

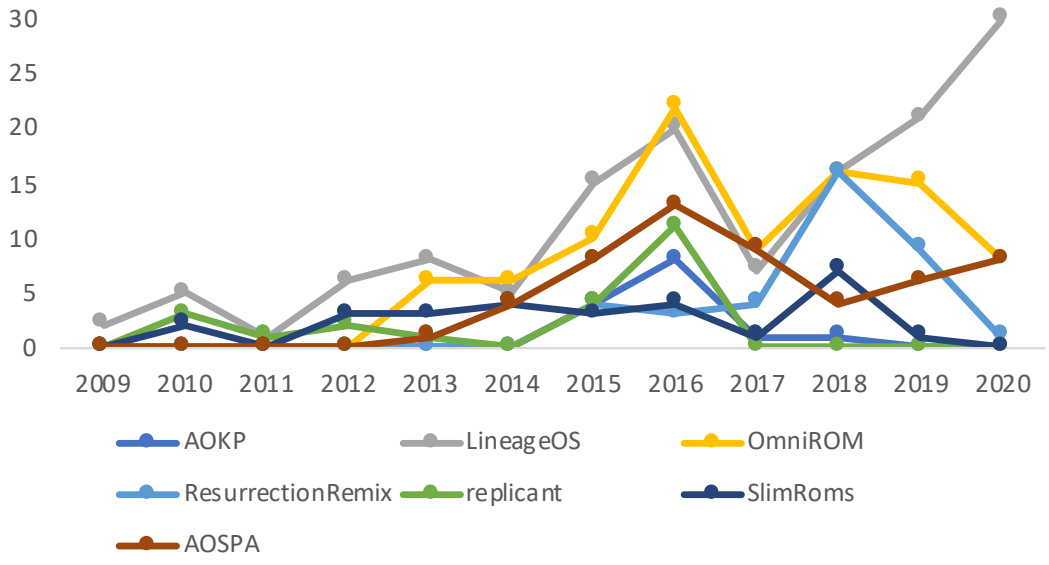


FIGURE 4.2: Merge commits with tags over time.

RQ2: ***What is the time lag between merge operations and corresponding commits in upstream repositories?***

Methodology: For every relevant merge commit identified in RQ1, we computed the time lag between the time the merge commit was performed and the time when the associated parent commit was made in the upstream repository. Considering the example in Listing 4.5, the analysis computed the time lag between commit `80d8b6467ec4` and commit `37a24f52e6be`.

Results: Table 4.3 reports our analysis results. Specifically, for each of the downstream projects, the table reports the number of merge commits considered (*Merge Commits*), the average time lag in days between the upstream commit associated with

a merge operation and the related downstream merge commit (*Time Lag*), and the average time difference in days between downstream-specific commits (*Downstream Commit*). Fig. 4.3 provides more details on the time lag and shows its distribution for all the projects. CRDROIDANDROID has an average time lag of 0.96 days and median time lag of 0.40 days and it is the shortest average time lag and median time lag across all projects. This very short time lag seem to indicate that **maintainers from this project closely follow the changes in the upstream repository**, and when the maintainers see relevant changes, they try to merge the changes immediately. RESURRECTIONREMIX, instead, is characterized by the longest average time lag and highest median time lag, which is equal to 37.60 days and 39.51 days, respectively. Excluding CRDROIDANDROID, all projects have an average lag time of more then 10 days and for most of the projects (five) the lag time is above 20 days. Moreover, some of the merge operations are even finished more than three months later, as shown in Fig. 4.3 for projects AOSPA and LINEAGEOS. The average time lag is in stark contrast with the time difference between downstream-specific commits, which for all projects is less than a day. This result reveals that **the time lag is a critical aspect characterizing downstream projects**. In fact, the long time lag might negatively impact downstream projects as they might see critical bug fixes and security updates only after several days or weeks. This situation might expose user devices running the customized OSs to security attacks.

RQ2 Findings

Most of the downstream projects take more than 20 days to bring changes from their corresponding upstream projects. Because security patches might not reach downstream projects in a timely manner, user devices running customized OSs might often be exposed to security vulnerabilities.

RQ3: *Do downstream developers experience conflicts when merging changes from upstream projects?*

Methodology: To identify conflicts, we processed the merge commits from RQ1, and for each commit, we use the information contained in the commit to (i) reset the state of the downstream repository to the parent commit from the downstream repository, (ii) reset the state of the upstream repository to the parent commit from

TABLE 4.3: Average lag time associated with merge commits.

Downstream PN	*Merge Commits*	*Time Lag (days)*	*Downstream Commit (days)*
AOKP	15	33.70	0.42
AOSPA	256	13.10	0.42
CRDROIDANDROID	351	0.96	0.28
LINEAGEOS	163	33.58	0.33
OMNIROM	251	18.25	0.52
REPLICANT	34	22.55	0.78
RESURRECTIONREMIX	45	37.60	0.24
SLIMROMS	33	31.20	0.67
Total/Average	1,148	23.87	0.46

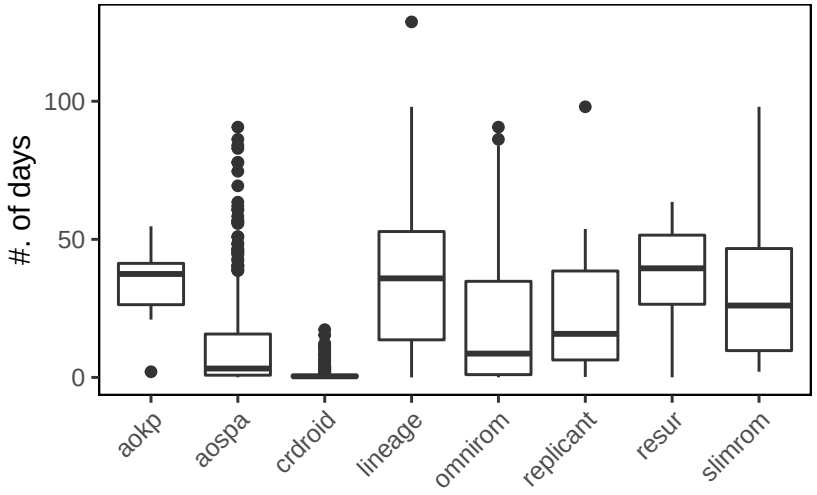


FIGURE 4.3: The distribution of merge time lag.

TABLE 4.4: Conflicts when merging changes from upstream.

Downstream PN	*Merge Commits*	*Conflicts*
AOKP	15	6 (40.0%)
AOSPA	256	138 (53.91%)
CRDROIDANDROID	351	78 (22.22%)
LINEAGEOS	163	70 (42.94%)
OMNIROM	251	128 (51.0%)
REPLICANT	34	18 (52.94%)
RESURRECTIONREMIX	45	16 (35.56%)
SLIMROMS	33	20 (60.61%)
Total	1,148	474 (41.29%)

the upstream repository, and (iii) perform the merge operation locally. In these steps, we reset the state of a repository using the `git reset --hard <commit-id>` command and we perform the merge operation by first adding the upstream repository as a remote repository of the downstream repository (using the `git remote add <remote-name> <upstream-repository-directory>` command), and then actually performing the merge

TABLE 4.5: Files affected by merge conflicts.

Downstream Project Name	*Files*						*Project Tot.*
	Java	aidl	cpp	xml	png	other	
AOKP	49	2	1	8	0	0	60
AOSPA	943	0	47	498	45	40	1,573
CRDROIDANDROID	116	5	1	23	2	2	149
LINEAGEOS	526	25	37	173	104	24	889
OMNIROM	934	32	35	454	53	42	1,550
REPLICANT	119	3	12	44	14	13	205
RESURRECTIONREMIX	204	8	8	46	0	4	272
SLIMROMS	179	15	11	55	139	2	401
File Type Total	3,070	90	152	1,301	357	127	5,099

operation (using the `git merge <remote-name>/<branch>` command). After we performed these operations for each of the merge commits, we counted the number of commits that lead to at least one merge conflict. To give an example, when we processed the merge commit reported in Listing 4.5, we reset the state of the downstream repository to commit `9b48a29cca52`, reset the state of the upstream repository to commit `37a24f52e6be`, and identified a conflict after we performed the merge operation.

Results: Table 4.4 reports the number of conflicts that we identified in our analysis. Specifically, for each of the downstream projects, the table reports the number of merge commits considered (column *Merge Commits*) and the number of commits affected by a conflict (column *Conflicts*). Across all projects, **41.3% of the total number of 1,148 commits are affected by a conflict** and all the projects are characterized by at least one conflict. The project with the highest percentage of commits affected by conflict is SLIMROMS while the project with the lowest percentage is CRDROIDANDROID. Even if CRDROIDANDROID has the lowest percentage, the project still encounters a considerable number of conflicts. These results show that **conflicts are an important aspect in the development of customized OSs**.

RQ3 Findings

Even though downstream projects do not experience a conflict in every merge operation, all of the projects experience conflicts, and most of the projects (six) experience a conflict 40% of the time.

RQ4: *What types of code entities are affected by the merge conflicts?*

TABLE 4.6: Characterization of files and code entities affected by merge conflicts.

	Merge Ops		Files	Code Entities										
Downstream PN	*Total*	*Conflicts*	*Java*	*Import*	*Class*	*Field*	*Constructor*	*Method*	*Enum*	*Class Comment*	*Field Comment*	*Constructor Comment*	*Method Comment*	*Comment*
AOKP	15	6 (40.0%)	49 (81.67%)	4	1	20	3	36	0	0	8	0	1	5
AOSPA	256	138 (59.95%)	943 (62.32%)	124	5	218	102	836	0	3	93	3	62	241
CRDROIDANDROID	351	78 (22.33%)	116 (77.85%)	30	1	30	9	90	0	1	17	0	1	14
LINEAGEOS	163	70 (42.94%)	526 (59.17%)	106	6	175	55	576	1	6	48	0	33	83
OMNIROM	251	128 (51.0%)	934 (60.26%)	156	8	223	104	876	0	5	93	3	64	178
REPLICANT	34	18 (52.94%)	119 (58.05%)	15	2	31	11	112	0	1	15	0	6	11
RESURRECTIONREMIX	45	16 (35.56%)	204 (75.00%)	55	1	75	36	217	0	0	17	0	8	38
SLIMROMS	33	20 (60.61%)	179 (44.64%)	34	1	71	16	198	1	0	18	0	15	39
Total	1,148	474 (41.29%)	3,070 (60.92%)	524	25	843	336	2,941	2	16	309	6	190	609

```
/**
  * <p>This can be used when the request is unnecessary
  * or will be superceeded by a request that
  * will soon be queued.
  *
  * @return the future id of the canceled request,
  * or {@link FillRequest#INVALID_REQUEST_ID} if
  * no {@link PendingFillRequest} was canceled.
  */
public CompletableFuture<Integer> cancelCurrentRequest(){
  return CompletableFuture.supplyAsync(() -> {
    if (isDestroyed()) { return INVALID_REQUEST_ID;}
    BasePendingRequest<RemoteFillService,
      IAutoFillService> canceledRequest =
      handleCancelPendingRequest();
      return canceledRequest instanceof
       PendingFillRequest ? ((PendingFillRequest)
         canceledRequest).mRequest.getId()
         : INVALID_REQUEST_ID; }, mHandler::post);
}

```

LISTING 4.6: Code snippet before update.

```
/**
  * <p>This can be used when the request is unnecessary
  * or will be superceeded by a request that
  * will soon be queued.
  *
  * @return the id of the canceled request,
  * or {@link FillRequest#INVALID_REQUEST_ID} if
  * no {@link FillRequest} was canceled.
  */
public int cancelCurrentRequest() {
  synchronized (mLock) {
    return mPendingFillRequest != null &&
    mPendingFillRequest.cancel(false) ?
      mPendingFillRequestId : INVALID_REQUEST_ID;
  }}

```

LISTING 4.7: Code snippet after update that led to a conflict.

Methodology: To identify code entities affected by the merge conflicts, we processed all the conflicts we identified in RQ3, and performed a two steps analysis. First, we

identified the files affected by conflicts and categorized the files based on their types. Second, for all the Java files (which are the predominant type of source code file in the part of the OS we analyzed), we also identified the types of code entities (e.g., import statements, class fields, method bodies, etc.) affected by the conflicts. In the analysis, we considered code entities so that we could characterize Java files in their entirety. To perform this second step, we map the conflict information to the abstract syntax tree (AST) of the Java files as they appear in the downstream repository before performing the merge operation.

Results: Tables 4.5 and 4.6 report the results of our analysis. Table 4.5 reports summary information describing the file types affected by the conflicts. Specifically, for each downstream project, the table reports the number of files of a certain file type that have at least one conflict. **Most of the time the conflicts involve Java files**. Specifically, for all projects, except for SLIMROMS, more than 55% of the files that have a conflict are Java files. The remaining portion of files contains aidl, cpp, xml, and png files, and also additional file types with included configuration and documentation files.

Table 4.6 provides details on the code entities affected by conflicts. In the table, column *Merge Ops* reports the total number of merge operations analyzed and the portion of operations leading to a conflict, column *Files* presents the number of Java files affected by at least one conflict, and column *Code Entities* details the number of code entities affected by a conflict divided by the entity type. The **entities most affected by conflicts are methods, followed by fields**. This result was expected as methods and fields are easy to leverage for extending downstream projects with new functionalities. It is worth noting that a considerable number of conflicts are caused by comments and import statements. This result suggests that the set of classes defined in the downstream projects is likely to be different as compared to upstream projects, which indicates that resolving the changes from the upstream projects might not be always an easy task. The last five columns in Table 4.6 summarize the number of conflicts caused by the comments associated with classes, fields, constructors, methods, and inline comments or stand-alone comments not specific to any code. This part of Table 4.6 shows **that not only executable code but also comments can cause conflicts** when merging changes from upstream to downstream repositories. After manually inspecting the conflicts associated with method comments, we identified that some of the conflicts are due

to changed method signatures and some are associated with semantic differences between the code in downstream and upstream repositories.

Listings 4.6 and 4.7 represent an example of a method comment that lead to a conflict. The method `cancelCurrentRequest` returns an integer value representing the identifier of a canceled request. Before resulting in a conflict, the method returned the future identifier through an integer wrapper class of `CompletableFuture`. After changing the code in the downstream repository, the method implementation was updated to return a primitive integer value. It is worth noting that the updated implementation leads to the change of the comment associated with the method. This change highlights that it might be possible to have code changes that are complex and involving semantic changes. Although the number of comments causing conflicts is smaller than the number of conflicts affecting executable code, **the difference in the comments suggests that some code changes are critical and hence have to be properly resolved by downstream projects**, like the ones highlighted in the motivating example of Section 4.2. If downstream developers do not properly handle updates to existing methods, user apps running on customized OSs might experience compatibility issues.

RQ4 Findings

The projects we considered experienced the majority of the conflicts in Java files. The most affected types of code entities are methods and fields. Our analysis also identified that code comments have conflicts, highlighting the possibility that the corresponding code conflicts might involve semantic changes.

RQ5: *What is the size of method-related conflicts?*

Methodology: In this research question, we compute the size of the conflicts affecting code methods to estimate the cost of analyzing and resolving the conflicts. We compute the size of the conflicts by looking at the number of lines and AST nodes involved in the conflicts. To compute the number of AST nodes involved in the conflicts, we leverage an AST parser [106] to analyze the portions of the Java source code files affected by the conflicts. Specifically, we first scan all the Java source code files affected by a conflict and then map AST nodes to the lines of the conflicts in the relevant files of the downstream and upstream projects. After that, we determine the size of the conflict by computing the number of nodes actually involved in lines affected by the conflicts.

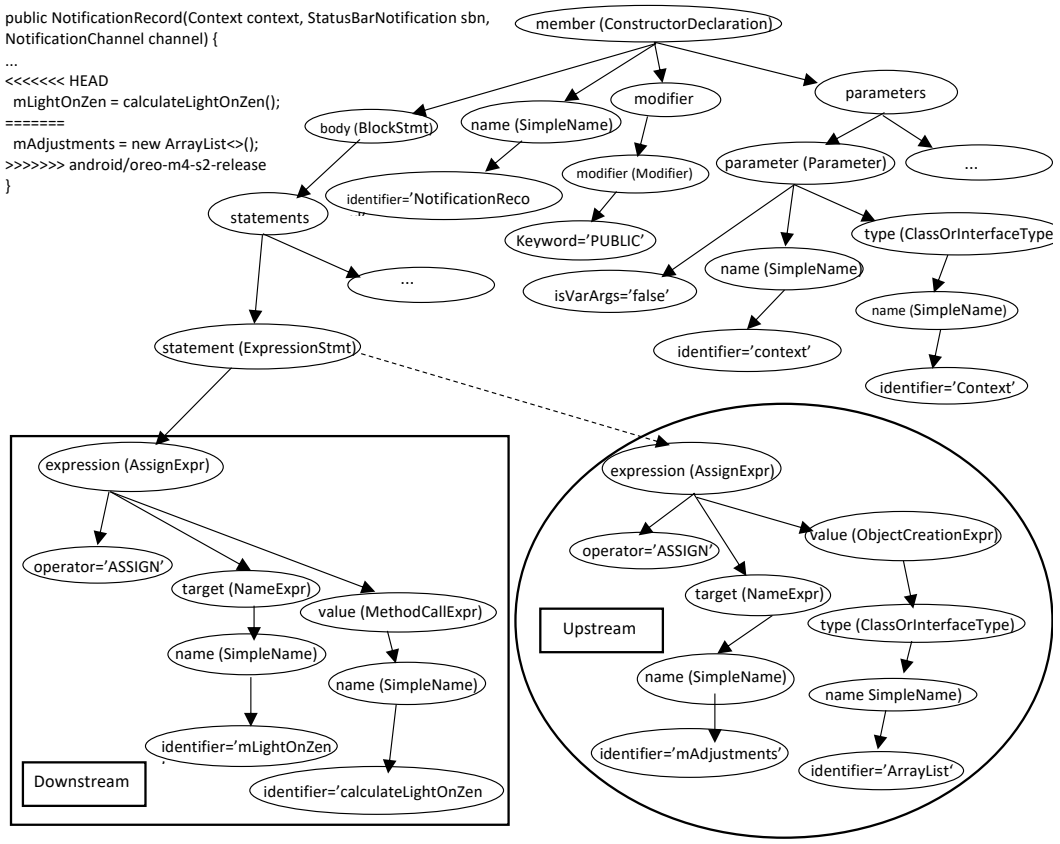


FIGURE 4.4: Sample AST.

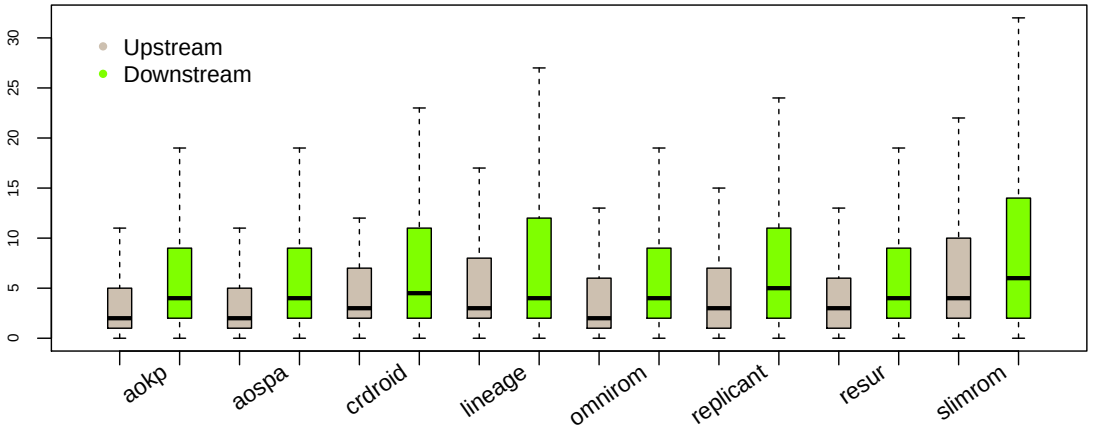


FIGURE 4.5: Lines affected by conflicts.

Fig. 4.4 provides an example of our analysis operates. The example depicts a portion of the AST of the sample code shown in the top-left part of the figure. The square and round parts are the parts identified by our analysis as the ones affected by the conflict. The square part shows the AST nodes of the code in downstream project (line below `<<<<<<< HEAD`) while the round part represents the code in the upstream repository (line below `=======`). In this RQ, we characterize the parts of the conflicts that are

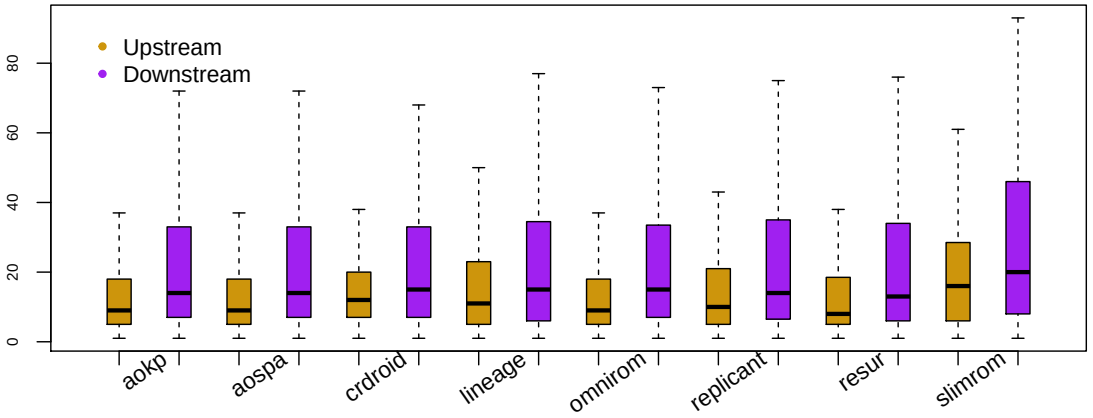


FIGURE 4.6: AST nodes affected by conflicts.

coming from both the downstream and upstream projects. We consider the parts from upstream because disregarding them might lead to compatibility issues.

Results: Fig. 4.5 presents the distribution of the number of code lines involved in the code blocks affected by a conflict in both the upstream and downstream projects. **The number of code lines from the downstream project is always larger than that of the upstream project**. After running a Mann–Whitney–Wilcoxon (MWW) test, we confirmed that the difference is significant. As revealed in our manual investigation, the rationale behind this difference is that developers of the downstream project have attempted to regularly update some of the methods, and such changes are often more frequent than that of the upstream project. This evidence experimentally shows that large and possible complex changes are indeed involved in method-related conflicts. Fig. 4.6 presents the distribution of the number of AST nodes involved in the conflicted code blocks in the upstream and downstream projects. Similarly to results presented in Fig. 4.5, **the size of changed AST nodes in the downstream project is larger than that of the upstream project**. This characteristic is confirmed as significant by the MWW test. This result confirms our previous findings that the changes in method-related conflicts could be complex.

RQ5 Findings

Conflicts affecting methods can involve a large number of AST nodes. Furthermore, the number of lines and AST nodes affected by conflicts are often higher in downstream projects than upstream projects. This result seems to indicate that developers significantly extend the functionality of upstream projects. When developers want to merge changes from upstream projects, they need to focus on a non-trivial portion of the code.

RQ6: *How often downstream developers change files in their projects when they experience a merge conflict?*

Methodology: With this research question, we are interested in understanding how often downstream developers change their projects when they experience a merge conflict. While answering this research question, we also look at how frequently developers disregard changes from upstream projects. When resolving a conflict, developers have three choices: (i) integrate upstream and downstream code changes by combining code entities from the two merged branches (*Integrate*), (ii) transform the downstream repository to have the same code as the upstream repository by removing the part of the conflict belonging to the downstream branch (*Use Upstream*), and (iii) keep the downstream code by removing the part of the conflict belonging to the upstream branch (*Use Downstream*). To answer the RQ, we classify conflict resolutions into one of the three categories by analyzing the ASTs of the downstream and upstream repositories before and after the merge operations.

Result: Table 4.7 summarizes the results of our analysis. The table reports the number of conflicts in the three categories we considered (*Integrate*, *Use Upstream*, and *Use Downstream*). The table also summarizes the total number of code entities characterizing the conflicts in the last column. The considered customizations have a significant number of code entities in the *Use Downstream* category. Specifically, the percentage of code entities in this category varies from 27.43% to 89.26%. This experimental result suggests that the **customizations of the Android framework stick to their changes during their evolution**. Although this characteristic is likely intended as customizations

TABLE 4.7: Code entities in the different resolution categories.

Downstream Project Name	*Integrate*	*Use Upstream*	*Use Downstream*	*Total*
AOKP	22	13	43 (55.13%)	78
AOSPA	809	354	524 (31.06%)	1,687
CRDROIDANDROID	53	21	119 (61.66%)	193
LINEAGEOS	375	195	519 (47.66%)	1,089
OMNIROM	802	439	469 (27.43%)	1,710
REPLICANT	77	44	83 (40.69%)	204
RESURRECTIONREMIX	33	15	399 (89.26%)	447
SLIMROMS	108	50	235 (59.80%)	393
Total	2,279	1,131	2,391 (41.22%)	5,801

aim to offer different versions of the Android OS, the differences (especially if they involve semantic changes) will exacerbate the fragmentation problem characterizing the Android ecosystem and cause compatibility issues to users.

RQ6 Findings

Downstream developers are required to resolve a large number of conflicts. This situation calls for automated techniques to help developers in the task. Furthermore, our results also show that in more than 40% of the cases, downstream developers disregard the portion of the conflict from the upstream repositories. This might lead developers to introduce compatibility issues.

RQ7: *Do developers of Android apps use methods affected by merge conflicts?*

Methodology: In this RQ, we are interested in investigating the use of conflict-affected methods in client Android apps. Since the implementation of the methods may be different between downstream projects and the official Android OS, there might be inconsistencies when customers are running apps containing these conflict-affected methods. The inconsistencies between these conflict-affected methods could lead to compatibility issues. There could be two kinds of compatibility issues. The first kind can appear when a downstream OS removes a method from the API that is instead present in the upstream OS. The second kind can appear when a method in the downstream OS has different semantics as compared to the same method in the upstream OS.

To empirically identify the extent to which conflict-affected methods are leveraged by real-world Android apps, we randomly select 1,000 apps from AndroZoo [107]. The apps in AndroidZoo are collected from real-world app markets such as the official Google Play store. For each of the selected app, we first disassemble it and scan its source code to collect all of its accessed methods from the Android API. We then compute the intersection between the used methods from the Android API and the conflict-affected methods identified in this work. If the result set is empty, we conclude that the app leverages no conflict-affected method. Otherwise, we identify that the app leverages x conflict-affected methods, where x is the size of the result set.

Results: 644 (or 64.4%) of 1,000 apps have accessed conflict-affected methods. The fact that over half of the randomly selected apps access such methods demonstrates that the potential impact of customization changes could be huge. Furthermore, analyzing the distribution of the number of accessed conflict-affected methods per app, we identified that **for most of the apps, there is more than one conflict-affected method accessed**, which further increases the possibility of encountering compatibility issues. Based on these results, we argue that the downstream project developers need to pay special attention to resolving the conflicts when merging updates from their upstream projects. Moreover, as a significant portion of conflicts has already been ignored by downstream developers (cf. the findings discovered in Section 4.4), certain compatibility issues might have already been introduced to the field.

RQ7 Findings

64.4% of randomly selected Android apps have accessed methods affected by conflicts. In more than half of the cases, the apps access more than one conflict-affected method. The use of these methods could result in compatibility issues and approaches to detect these potential issues are needed.

4.5 Discussion

Continued research on techniques to assist updates from upstream is needed. As most of the downstream projects that we considered encounter more than 40% merge conflicts and resolving conflicts often involves considering a large number of code entities (RQ5), we argue that there is a strong need to continue devising automated approaches to help complete those merge-operations. This is especially true for helping

customized Android frameworks keep the evolution pace with the official Android framework, which is one of the most actively maintained and largest open-source projects. As argued by Owhadi-Kareshk et al., automated predictors could be also leveraged to predict potential merge conflicts, as a pre-filtering step for speculative merging [108]. Additionally, given the variance we identified in the merge practices across different projects, we believe that it is necessary to extend our results by performing studies that involve the developers of the customized projects to further understand constraints in their merge practices and even more suitably guide the creation of techniques and tools to merge from upstream.

Merge changes from upstream as soon as possible. Downstream project developers should merge changes as soon as possible, since the less time lag before merging from upstream, the fewer merge conflicts they may encounter. Since downstream projects always need to be synced up with upstream to bring in features, bug fixes, etc., it is good to have routine merging approaches from upstream, which would relieve developers' burdens in resolving the merge conflicts. It could be even better if automated approaches can be proposed to support such timely merge strategies.

Automated techniques are needed to help developers handling comments conflicts. Downstream customizations have a large number of conflicts appearing in inline or stand-alone comments (e.g., comments in the middle of a class not directly associated with code). Since comments are of great importance for developers of these downstream projects, the conflicts associated with inline or stand-alone comments can be time-consuming to resolve as developers need to first relate the changes in the conflict to specific code sections. To improve the development of customized OSs, automated techniques are needed to relate comments changes to code changes. These techniques could leverage the information contained in the version control system and use static analyses combined to natural language processing to relate changes to code sections automatically.

Merge details are needed to aid the evolution of the customized OSs. In the study, we identified that downstream developers do not follow a systematic approach for merging updates from upstream projects. Additionally, in our manual inspections, we identified that developers rarely use detailed commit messages to described the merge operations they performed. Because of this situation, different developers in the same

project might not be aware of semantic changes, bug fixes, or security patches applied (and not applied) by other developers, and this might have negative consequences on their development activities. We suggest that developers of customized OSs follow merging strategies agreed upon in the projects or provide detail commit messages in their merge operations describing the changes applied to the project. To this end, automated approaches could suggests commit summaries based on the changes made in the merge operations. This advice also extends to files other than source code. For example, we identified a large number of xml files affected by merge conflicts, and those files can affect the UI or default configurations of the OS.

Mitigated compatibility issues. As revealed in our experimental study, developers of downstream projects have ignored some of the changes from upstream projects. Some of the changes, e.g., relevant to fixing bugs or augmenting features, may cause semantic differences between the two frameworks, which are supposed to support the execution of the same apps. Subsequently, those changes could lead to potential compatibility issues, which are hard to identify and hard to account for app developers. We argue that automated approaches are also needed to identify and mitigate instances of this situation, e.g., by (i) detecting semantic deviations from upstream projects, (ii) helping downstream project developers to better merge the changes of their upstream projects, or (iii) by helping app developers protect (or avoid) the usages of methods that may cause compatibility issues.

4.6 Threats to validity

The primary **external threat to validity** of this study lies in the choice of downstream projects. In this study, we have only analyzed eight different open-source downstream projects and the results might not generalize to other projects. To mitigate this threat, we conducted experiments on all of the downstream projects that we could readily get and these projects include different customizations of the Android OS. We also believe that an interesting direction for future work could focus on extending our results by searching and considering customizations of the Android OS that are available on GitHub. Furthermore, our research might not generalize to close-sourced projects such as MIUI [109] or OnePlus [110]. However, after inspecting the release messages of

those projects, we noticed that those projects have update patterns that are in line with the ones we identified in our study. Although encouraging on the possibility that the results might generalize, additional studies based on those projects are needed to further understand upstream merge practices in the context of Android.

The main **threat to construct validity** resides in the possibility that the implementation of our experimental scripts and tools to do the analysis might contain errors. To mitigate this threat, we extensively inspected the results of the study manually. For example, we determine if the conflict is handled by ignoring the code snippet from upstream projects through manually checking whether the code snippet exists in the Java files associated with the completed merge operation. The authors also cross-validated the results.

4.7 Related work

There are other works that have studied merge conflicts [50–52, 111–114] in divergent projects/branches. In this section, we describe related work closely related to ours.

Mens [50] performed a survey to investigate a wide range of different merge techniques, from the initial pure textual merging to the more powerful approaches taking syntax and semantics into account. Various merge techniques can be classified into different categories from different perspectives. In addition, the author in the paper compares these general merge techniques based on many different important criteria. Different from the survey comparing several different merge techniques, we focus on the performed textual merge operations in different customized Android projects.

Mahmoudi et al. [51] carried out an empirical study to determine the details of the updates between Lineage OS and the update of the official Android. They found that 83% of subsystems updated in LineageOS is also modified in official Android and that 56% of the overlap modifications in LineageOS can be automatically safely merged into the next new release version. While they only focus on eight different update problems of the details of the changes applied in LineageOS versus those in Android, we highlight the merge conflicts during the evolution of Android but not limited to LinegeOS, which means they analyze the final result of the evolution but we highlight the merges during the whole evolution process.

Cavalcanti et al. [115] and Accioly et al. [116] carried out a comprehensive study on semistructured merge on the evolution of large-scale open-source Java projects. Accioly et al. conducted an empirical study on large-scale open-source Java projects to figure out the characteristics of merge conflicts during merge by a semistructured merge tool over 70,047 merges from 123 Github Java projects. They conclude that the merge conflicts usually involve more than two developers, which means they need to understand different branches developed by different developers to successfully merge. Cavalcanti et al. did a comprehensive study over more than 30,000 merges from 50 open-source projects. They found that some of the conflicts induced by unstructured merge can be auto-removed and also the merge conflicts are easier to analyze and resolve. Therefore, they proposed an advanced semistructured merge tool combining both approaches when merging and the experiment results indicate the tool is promising in reducing the false positive merge conflicts. Different from these study, we did not involve semistructured merge but only focused on unstructured merge operation and their conflicts resolution.

Ghiotto et al. [117] performed an extensive comprehensive study on the merge conflicts in the histories of 2,731 open source Java projects. They found that 40 percent of the failed merges have a single conflicting chunk and 90 percent have 10 or fewer. The majority of the conflicting chunks have up to 50 LOC (Line Of Code) in each version and the method invocation is the most frequent language construct. Among different constructs in merge conflicts, they revealed that if statement, method invocation are of the most difficulties. We also did an extensive analysis of merge conflicts in the evolution of Android customizations. However, we only focus on the merge operations bringing in updates from upstream projects without taking branch or fix merges into consideration, which would have some negative effect on the synchronization merges from upstream projects.

Shen et al. [118] proposed a new merge approach called IntelliMerge. The approach provides a refactoring-aware merging algorithm for Java programs. The evaluation of the approach is based on 10 Java projects and it reduces the number of merge conflicts as compared to related work chosen as the baseline. Nishimura et al. [119] also present a tool, MergeHelper, to help in merging independent development from different developers by replaying the detailed code changes related to the conflict class members.

Lamothe et al. [120] did an extensive systematic literature review of API evolution including Android and other Java-based APIs. Inspired by the approaches in existing literature [121, 122], they utilized a systematic approach to initiate a survey related to API evolution. By collecting online literature from five well-known technical publishers, they summarized different challenges raised by different researchers and concluded with three dominant challenges, including API changes pinpointing, benchmark creation for the analysis of API evolution, and understanding the impact of API evolution.

4.8 Summary

In this paper, we presented an empirical study that investigated how developers of customized versions of the Android OS update their projects to merge changes from the main version of the Android OS. In our study, we analyzed the merge operations from eight open-source customized Android OS projects and identified how these operations affect a randomly selected sample of 1,000 apps. Our results show that the developers performed a small percentage of the possible updates, the merge operations often lead to conflicts, and a large percentage of the apps considered use methods affected by conflicts. The large number of conflicts identified and the low percentage of merge operations performed motivate further research on automated or semi-automated techniques to support the merge operation task. Furthermore, the high number of conflicts and the high percentage of apps using methods affected by conflicts indicate that these apps might experience compatibility issues, motivating further research in helping app developers handle this type of issues. As future research, we plan to present our findings to developers that work on customizations of the Android OS to determine changes they find most challenging to merge. We then plan to develop automated techniques to help developers correctly perform merge operations more frequently, efficiently and effectively.

Chapter 5

Compatibility Issues Detection

Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. "Automatically Detecting Incompatible Android APIs." ACM Transactions on Software Engineering and Methodology (TOSEM), 2023. Accepted

5.1 Introduction

Fragmentation has been a severe problem for the Android ecosystem for years. "Fragmentation" refers to the fact that there is a massive number of Android devices manufactured by different companies running different Android operating system versions, including both official and customized ones. This introduces inconsistencies in that certain apps can only function properly on devices running specific Android versions with certain device features (i.e., the apps crash or don't work properly on different devices), leading to so-called *app compatibility issues*, short for *compatibility issues* in the paper, which include forward compatibility issues and backward compatibility issues [14]. The typical forward compatibility issue refers to API deletion in newer Android releases. An Android app will not function properly in newer Android releases due to the invocations of such removed APIs. On the contrary, the backward compatibility issue implies new APIs have been added in newer releases. An Android app will not function properly on older releases if its implementation utilizes such added APIs.

Compatibility issues have been considered one of the most severe problems in the rapidly and constantly evolving Android ecosystem. They can be induced by incorrect invocations of APIs, which we call API-related, or incorrect permission acquisition, non API-related [22]. On the one hand, they negatively impact the users' experience, as apps with compatibility issues may not be able to install on users' devices or may crash at runtime. On the other hand, they also increase the difficulties of developing apps. The vast number of device-Android version combinations create many technical complexities for developers and testers, which are non-trivial and yet expensive to resolve without a proper infrastructure in place.

To address these issues, we focus on API-related compatibility issues. There has been a significant amount of research in analyzing the compatibility issues induced by incorrect API invocations in Android apps. In the area of static analysis, researchers have proposed various automated approaches to pinpointing one of the most common compatibility issues: evolution-induced compatibility issues,. These refer to Android apps implementing and functioning properly on certain versions of official Android systems but not on others as of provision of certain Android APIs on these systems. For example, Li et al. [14] have designed and implemented a prototype tool called CiD that mines the evolution of the official Android framework codebase to locate evolution-induced incompatible Android APIs, i.e., new methods introduced in or existing methods being removed from the latest framework versions, and detects compatibility issues via static analysis approach based on these identified incompatible APIs. He et al. [10] also introduced a detection tool called IctApiFinder to detect evolution-induced compatibility issues. They first harvested incompatible APIs by extracting APIs from the released Android SDKs from version API level 4 to API level 27 and determining if there are Android APIs newly added or deleted, called incompatible APIs, in a newer released version, and then detected compatibility issues (induced by the incorrect invocations of these incompatible APIs) with their static analysis implementation. Wei et al. [23] have proposed a prototype tool called Pivot for characterizing device-specific incompatible APIs, e.g., APIs that are available for certain devices but not for others. Huang et al. [66] deeply explored the compatibility issues caused by the evolution of callback APIs, and proposed CIDER, which utilizes a graph-based model to detect API callback compatibility issues. We refer to different types of compatibility issues according to the

incompatible APIs causing such issues, such as callback methods induced compatibility issues [66] and evolution-induced compatibility issues [10, 14], etc.

Although related work has attempted to tackle compatibility issues, it has not yet been entirely clear what the strengths and weaknesses of state-of-the-art tools are and to what extent they are able to identify different types of compatibility issues in real-world Android apps. Furthermore, it is also unknown to which extent we can reproduce experimental results from related work and how well each of the tools compares with each other in terms of detecting different kinds of compatibility issues. In this work, we formulate these concerns into three research questions that we aim to answer through empirical evidence and experimental results. Our three research questions are summarized as follows:

- **RQ1: What is the state-of-the-art tool performance in Android compatibility issues detection?**

  We answer this question through a systematic literature review, aiming to identify the primary studies relevant to statically detecting Android app compatibility issues. Our review identified nine primary publications that have proposed automated approaches for characterizing Android app compatibility issues. After careful analysis, we summarize five identified types of API-induced compatibility issues: evolution-induced (method), evolution-induced (field), device-specific (method), device-specific (field), and override/callback methods. Unfortunately, none of the existing approaches can tackle all five types of API-induced compatibility issues. The most recent, ACID [24], can only handle three out of the aforementioned five types of compatibility issues.

- **RQ2: Can we replicate the experimental results yielded by state-of-the-art tools targeting compatibility issue detection?**

  Replicability studies are regarded as an essential method to confirm the reliability of existing research (including both experiments and datasets). They are becoming an important focus in the software engineering community. To answer our second research question, we confirm the reliability of existing state-of-the-art compatibility issues detection tools by reproducing their experimental results with their original datasets. Our experimental results show that the majority of

these study results can indeed be reproduced. The remaining small number of inconsistent results (yielded by IctApiFinder and FicFinder) are mainly caused by updates of the tools (such as dependency fixes) and apps (due to the unrecorded Github versions of the apps).

- **RQ3: How well do the tools in issue detection compare with each other?**

  To answer this question we apply selected state-of-the-art tools on two common sets of benchmark apps: (1) 65 apps used by the authors of selected tools and (2) 645 apps selected from the AndroidCompass dataset [123]. Our experimental results show that (1) compatibility issues detection approaches that achieve their purpose via systematically harvested incompatible API rules (such as CiD and IctApiFinder) can identify significantly more issues than those having their rules summarized manually, and (2) the intersection among the results reported by the selected tools is relatively small.

Our empirical study shows that typical compatibility issue detection involves two separate steps, including incompatible APIs gathering and compatibility issue determination (induced by such incompatible APIs), which are all achieved by the detection tools, CiD, IctApiFinder, CIDER, and FicFinder. To be more specific, compatibility issues in our paper refer to the invocations of incompatible APIs without guard checks for specific Android versions and/or devices. However, we empirically find that our community has not yet defined systematic approaches for identifying incompatible Android APIs, including evolution-induced APIs and device-specific ones, and many incompatible APIs have been overlooked. Even for the ones that can be detected, the intersection among the results given by the state-of-the-art tools, which mainly focus on detecting compatibility issues induced by the evolution of official Android APIs, is also relatively small [1]. Since incompatible APIs play a crucial role in compatibility issue detection, we go beyond the state-of-the-art approaches by presenting a new prototype tool for harvesting incompatible APIs. It is worth mentioning that the identified detection tools (CiD, IctApiFinder, CIDER, and FicFinder, except for Pivot) are all issue detection artifacts. They not only gather incompatible APIs but also pinpoint compatibility issues in Android apps. However, the approaches of harvesting incompatible APIs utilized by these issue detection

[1] It is a rough comparison. We will discuss this better in threats to validity.

tools and Pivot are not systematic or evolution-induced oriented. Our tool, named *AndroMevol*, advances the approach to gathering incompatible APIs by taking into account both the fast evolution of the official Android framework and the system customizations made by different device manufacturers. To evaluate the effectiveness of the approach, we further propose to answer two more research questions:

- **RQ4:** How well does *AndroMevol* perform in automatically identifying incompatible Android APIs?

  We selected five popular Android brands (i.e., Huawei, Xiaomi, Oneplus, OPPO, and Samsung) and evaluated if *AndroMevol* is capable of automatically generating a list of incompatible Android APIs, including methods and fields, for helping researchers and developers pinpoint potential compatibility issues in Android apps. Using the full history of the official Android framework and 52 customized Android frameworks extracted from devices of popular brands. Our *AndroMevol* approach was able to report 397,678 incompatible APIs that do not exist in all the considered platform versions.

- **RQ5:** How is *AndroMevol* compared with state-of-the-art approaches targeting automated collection of incompatible APIs?

  To answer this research question, we compare our tool with other state-of-the-art approaches, CiD, CIDER, IctAPIFinder, FicFinder, and Pivot, in harvesting incompatible APIs. Compared with these selected tools, our *AndroMevol* approach can pinpoint many more incompatible APIs, enhancing the ability to detect compatibility issues in Android Apps for the state-of-the-art issue detection tools.

This work extends our conference paper [124] by proposing to the community a new prototype tool called *AndroMevol* to systematically harvest incompatible APIs. Our experimental results show that *AndroMevol* is effective in harvesting incompatible Android APIs. It also outperforms state-of-the-art approaches by harvesting at least eight times more incompatible APIs, including 195,883 APIs that have never been reported previously. Furthermore, we have updated the Discussion and Related Work sections to cope with the aforementioned changes. The source code and datasets are all made publicly available in our artifact package via the following link:

https://github.com/MobileSE/AndroMevol

In summary, the main contributions of this paper are:

- A systematic literature review across 19 top-tier venues to apprehend the status quo of compatibility issue detection. We carefully read the targeted nine relevant papers from the selected top venues, summarized five types of compatibility issues, and identified seven state-of-the-art detection tools to pinpoint such issues.
- A comprehensive comparative study to compare the ability of each identified detection tool. As the detection tool ACID and ACRYL cannot be replicated and Pivot is developed only for incompatible APIs collection, we could successfully replicate four detection tools, including CiD, IctApiFinder, CIDER, and FicFinder. To measure the ability of the issue detection, we run the detection tools against two datasets. One is the original dataset published along with the detection tools and the other is the dataset AndroidCompass, which contains open-source Android projects collected from GitHub. Our evaluations on the original dataset demonstrated that CiD outperforms the other detection tools by detecting more than one and ten times the number of compatibility issues pinpointed by IctApiFinder and FicFinder, respectively. We also found that typical issue detection approaches involve two separate steps, including incompatible APIs identification and compatibility issue determination, and incompatible APIs play a crucial role in issue detection. To boost the ability of the issue detection of the tools, a complete set of incompatible APIs is necessary.
- A systematic approach to harvesting incompatible APIs, including methods and fields, among five popular OS brands, including Huawei, Xiaomi, Oneplus, OPPO, and Samsung alongside AOSP. We proposed the new prototype tool called *AndroMevol* to harvest incompatible APIs from the official Android frameworks and 52 customized Android frameworks extracted from the released ROMs of the five popular brands. With the *AndroMevol*, we could collect a total of 397,678 incompatible APIs including 195,883 previously untouched ones, which could be accessed by Android apps inducing compatibility issues. Compared to other API harvest tools, *AndroMevol* performs better by harvesting at least eight times more incompatible APIs.

5.2 State-of-The-Art App Compatibility Detection Approaches (RQ1)

We performed a systematic literature review to understand the current state-of-the-art in Android app compatibility analysis approaches and available tools and datasets.

5.2.1 Literature Review

Figure 5.1 illustrates the working processes of our literature review, summarized based on the guidelines provided by Kitchenham et al. [121] and Brereton et al. [125], as well as lessons learned from our own recent SLR practices [126–129].

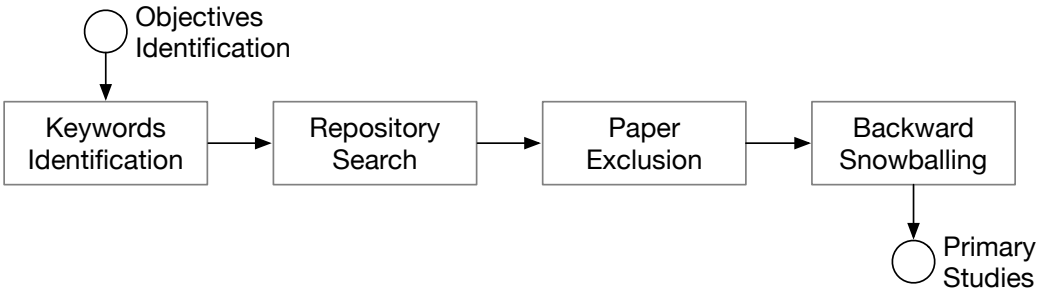


FIGURE 5.1: Overview of the literature review process.

Keyword Identification. To understand the status quo of incompatible app analyses, we use a set of keywords to search for key relevant publications in popular repositories. The keywords we leveraged are essentially made up of two groups (i.e., G1 and G2). Each group contains several keywords. The search string is then formed as a combination, i.e., g1 AND g2, where g1 and g2 are formed each as a disjunction of the keywords respectively from groups G1 and G2.

$$G1 : android, mobile, *phone*$$

$$G2 : *compati*, deprecat*, issue*, evolution$$

Repository Search. To focus our search, we applied these keywords on all the CORE[2] A/A* ranked venues. This keeps the review process lightweight while ensuring that important related works are not missed. In the software engineering field (i.e., containing 'software' keyword in the venue title and falling in the following fields of research code:

[2] https://www.core.edu.au/home

0803 for journals and 4612 for conferences), as summarized in Table 5.1, there are 19 venues (5 journals and 14 conferences) ranked as A/A* by CORE. These are all the top SE publication venues where high-quality Android app compatibility detection work is typically published. We then went through these 19 venues one by one and applied the aforementioned keywords to search for relevant publications. Eventually, we were able to locate 44 publications across 13 venues (i.e., there is no relevant paper identified in 6 of the venues).

TABLE 5.1: CORE A/A* ranked software engineering venues.

Type	Source	Venues
Journals	CORE2020	TOSEM, TSE, EMSE, JSS, IST
Conferences	CORE2021	ASE, ESEC/FSE, ICSE, EASE, ECSA, ISSRE, ESEM, ICSME, MSR, ICSA, SANER, SEAMS, ICST, ISSTA

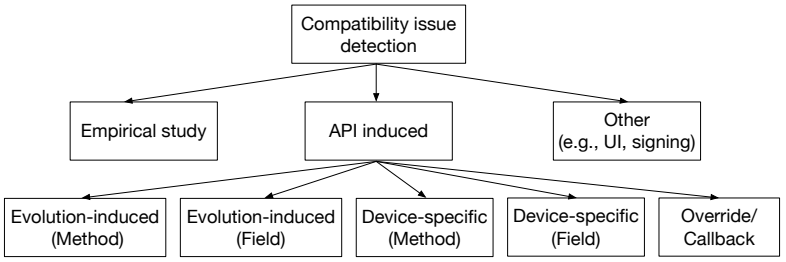


FIGURE 5.2: The category of the papers targeting compatibility issues on the Android platform.

Paper Exclusion. As we aimed at collecting as many relevant papers as possible, we have simply considered all the returned results. However, not every paper is related to automated Android app compatibility issue detection. We there go one step further to read the abstract (and full content if needed) of the obtained papers to only retain the closely related ones by applying the following exclusion criteria: (1) Short papers (i.e., less than six pages in double-column format or 11 pages in single-column format) are excluded. (2) Papers targeting non-Android mobile devices are excluded. (3) Papers targeting Android but that do not concern compatibility issues are excluded. (4) Papers targeting Android compatibility issues but that do not concern API-induced ones are excluded (categorized as Other in Figure 5.2). For example, the work presented by Ki et al. [130], which proposes an automated testing framework for Android apps named Mimic for characterizing UI compatibility issues, is excluded. Another work presented by Wang et al. [131], which has discussed a type of app signing compatibility issue introduced by unsupported digest/signature algorithms, is also excluded. (5) Papers

targeting Android compatibility issues but that do not introduce automated approaches to detect or resolve them are excluded (categorized as Empricial Study in Figure 5.2). For example, Nielebock et al. [123] contribute an Android compatibility check dataset named AndroidCompass, which comprises changes to compatibility checks in the version histories of the Android projects. Cai et al. [22] conduct a large-scale study of compatibility issues based on Android apps developed over the past eight years to comprehend the symptoms and root causes. These papers do not introduce a prototype tool to detect compatibility issues in Android apps and hence are excluded. After applying these exclusion criteria, there are nine papers retained that are closely related to automated incompatible Android API detection.

Backward Snowballing. Based on the papers identified in the previous steps, we conducted a backward snowballing approach to ensure that important closely related papers (e.g., with titles not matching our search string or published outside of the selected 19 venues) are not missed by our lightweight literature review. For each paper we carefully read the related work part and attempted to find cited papers that are closely related to our study but have not yet been included. This process did not help us identify any new papers, suggesting that the keywords and venues that we selected to search for relevant publications are indeed the most relevant ones.

TABLE 5.2: Full List of Collected and Examined Papers.

Tool/Reference	Year	Venue	Tool availability
ACID[24]	2021	SANER	Available [132]
ACRYL (extension)[72]	2020	EMSE	Open Source [133]
ACRYL[20]	2019	MSR	Open Source [133]
Pivot[23]	2019	ICSE	Available [134]
CiD[14]	2018	ISSTA	Open Source [135]
IctApiFinder[10]	2018	ASE	Open Source [136]
CIDER[66]	2018	ASE	Available [137]
FicFinder (extension)[138]	2018	TSE	Available [139]
FicFinder[19]	2016	ASE	Available [139]

5.2.2 Result

In total, our Systematic Literature Review (SLR) search process identified nine relevant papers (hereinafter referred to as primary studies, which are listed in the first column of Table 5.2. The nine papers are collected from seven venues with publication dates

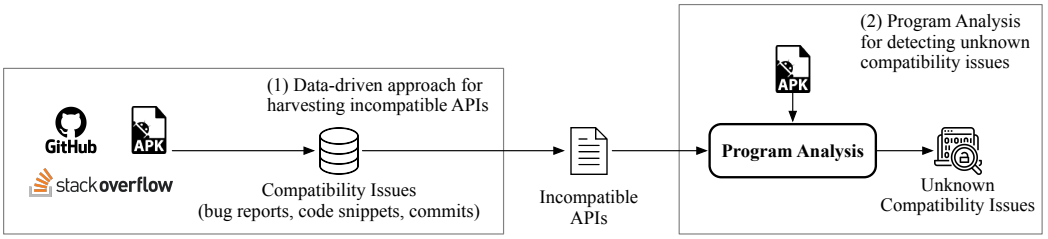


FIGURE 5.3: The typical working process of detecting Android compatibility issues.

ranging from 2016 to 2021 (cf. second and third columns in Table 5.2). The last column describes the availability of these tools. Some of them are open-sourced, while others are published as executable files on the associated papers' websites.

5.2.3 State-of-the-Art App Compatibility Analysis Approaches

After identifying the primary publications, we carefully read their full papers to understand how each of their automated compatibility issues detection approaches is implemented. We then summarize the common working process taken by those approaches to detect Android compatibility issues.

As shown in Figure 5.3, the objective is often achieved via two steps: (1) data-driven approach for harvesting incompatible APIs and (2) program analysis for detecting unknown compatibility issues. The output of the first step will be a list of incompatible APIs, which will be taken as input to the second step. With the two typical steps of the working process of compatibility issue detection, we summarized the collected tools as in Table 5.3. The second and third columns describe incompatible APIs collection and issue detection per se separately. CIDER and FicFinder only support issue detection while Pivot only focuses on incompatible APIs collection regardless of publicly available ones or conventionally restricted ones. The remaining tools are working as a whole supporting both APIs harvesting and issue detection.

After carefully reading their full content, we understand that compatibility issues stem from API inconsistencies that are induced by the evolution of Android OS per se and different OS customizations. We, therefore, categorize compatibility issues into five types as described in Figure 5.2 according to the different types of underlying inconsistent APIs. For each of the considered tools, we further summarize and list its capabilities in Table 5.4. It is worth noting that FicFinder does have the ability to detect device-specific

TABLE 5.3: Working Process Support of Tools.

Tool/Reference	API Harvest	Issue Detection
ACID[24]	✓	✓
ACRYL (extension)[72]	✓	✓
ACRYL[20]	✓	✓
Pivot[23]	✓	✗
CiD[14]	✓	✓
IctApiFinder[10]	✓	✓
CIDER[66]	✗	✓
FicFinder (extension)[138]	✗	✓
FicFinder[19]	✗	✓

methods-induced compatibility issues but the incompatible APIs leveraged by the artifact are all evolution-induced ones. Therefore, we assume the artifact FicFinder could only detect evolution-induced compatibility issues in the following discussion. Columns 2-6 describe the detection of the five types of compatibility issues, which are further detailed with concrete examples as follows.

TABLE 5.4: Examination results of the approaches proposed in the retained primary studies.

Tool/Reference	Evolution-induced (Method)	Evolution-induced (Field)	Device-specific (Method)	Device-specific (Field)	Override/ Callback	Systematic (Sound)	Fully automatic[2]
ACID[24]	✓	✓[1]	✗	✗	✓	✓	✓
ACRYL (extension)[72]	✓	✗	✗	✗	✗	✗	✓
ACRYL[20]	✓	✗	✗	✗	✗	✗	✓
Pivot[23]	✓	✗	✓	✗	✗	✗	✓
CiD[14]	✓	✗	✗	✗	✗	✓	✓
IctApiFinder[10]	✓	✓[1]	✗	✗	✗	✓	✓
CIDER[66]	✗	✗	✗	✗	✓	✗	✗
FicFinder (extension)[138]	✓	✗	✓	✗	✗	✗	✗
FicFinder[19]	✓	✗	✓	✗	✗	✗	✗

[1] Only mentioned but not illustrated in detail.
[2] There is no human involvement in the core process, e.g., the learning/knowledge collection phase.

Evolution-induced (Method): The signatures [3] of some public methods are altered (i.e., removed, newly added, or parameter type changes, etc.) during the evolution of the framework. Figure 5.4 demonstrates such an example, for which the code snippet is initially reported in [10], where statements beginning with the + signs indicate a possible fix for this incompatibility. The API *startDrag()* called on Line 7 is introduced into SDK after level 11. However, the minSdkVersion of this app is set to 10. Consequently, if not protected with the "if-else" block, a "NoSuchMethodError" exception will be thrown, leading to crashes on devices running SDK version 10.

Evolution-induced (Field): During the evolution of the framework, the signatures of some publicly accessible fields could also be altered (i.e., removed or newly added). Unfortunately, apart from [10] and [24], none of the other papers discusses such issues.

[3]The method signature in our research and current related work refers to the combinations of the method return type, method name, and method parameters type list.

```
1   public class MainActivity extends
        Activity{
2    private TextView mView;
3    protected void onCreate(Bundle
        bundle) { ...
4  +   if(Build.VERSION.SDK_INT >= 24)
5  +     wrapper(mView, c, s, null, i);
6  +   else
7       mView.startDrag(c, s, null, i);
8    }
9  + private wrapper(View v, ClipData c, ...)
        {
10 +   v.startDragAndDrop(c, s, o, i);
11 + }
12   }
13
```

FIGURE 5.4: Example 1: Evolution-induced(Method)

```
1   public static Bitmap getCachedArt(final
        Context context,final Song song){
2    ...
3    Options options=new Options();
4    options.inDither=false;
5    options.inPreferredConfig=ARGB_8888;
6    ...
7   }
8
```

FIGURE 5.5: Example 2: Evolution-induced(Field)

```
1    Camera mCamera = Camera.open();
2    Camera.Parameters params =
        mCamera.getParameters();
3    .....
4  + if (android.os.Build.MODEL.equals("
        Nexus 4") {
5  +  params.setRecordingHint(true);
6  + }
7    .....
8    mCamera.setParameters(params);
9    mCamera.startPreview();
10
```

FIGURE 5.6: Example 3: Device-specific(Method)

```
1   private static HttpClient getNewHttpClient() {
2    ...
3    sf.setHostnameVerifier(SSLSocketFactory.
        ALLOW_ALL_HOSTNAME_VERIFIER);
4    ...
5   }
6
```

FIGURE 5.7: Example 4: Device-specific(Field)

```
1    public void onAttach(Context context) {
2     super.onAttach(context);
3  -  mActivity = (BrowserActivity) context;
4  -  .....
5  +  attachActivity((BrowserActivity) context);
6    }
7  + public void onAttach(Activity activity) {
8  +  super.onAttach(activity);
9  +  if (Build.VERSION.SDK_INT < 23) {
10 +    attachActivity((BrowserActivity) activity);
11 +  }
12 + }
13 + private void attachActivity(BrowserActivity activity) {
14 +  mActivity = activity;
15 +  .....
16 + }
17
```

FIGURE 5.8: Example 5: Override/Callback

FIGURE 5.9: Code Examples

Moreover, no relevant examples are given in all the research papers. Then we use an example that we discovered throughout our research. There is an evolution-induced issue with a field called ”BitmapFactory.Options.inDither” at Line 4 of Figure 5.5. It’s supported by API Levels 1 through 23, however, since API Level 24, it’s been deprecated, creating compatibility issues when an app sets a target SDK version equal to or greater than 24.

Device-specific (Method): Due to the customization of smartphone manufacturers, some APIs only work on some devices but not on others. Figure 5.6 demonstrates such an example, originally reported by Wei et al. [23]. Only if the result of the conditional statement for checking the device identifier according to “Nexus” is true, that is, the corresponding app is indeed running on “Nexus”, the API *setRecordingHint()* on Line 5 will be executed.

Device-specific (Field): Similar to evolution-induced compatibility issues, the customization of Android frameworks can also introduce incompatible fields (i.e., exist in some devices but not in others), referred to as device-specific fields. No code example is provided in our reviewed primary papers, similar to Evolution-induced (Field). We then take the example of "<org.apache.http.conn.ssl.SSL-SocketFactory: org.apache.http.conn.ssl.X509HostnameVerifier ALLOW_ALL_HOSTNAME_VERIFIER>", as shown in Figure 5.7. According to our analysis results, this field is not supported by OPPO smartphones in the SDK of API Level 26, which account for more than 10% of global smartphone shipments [140]. If an app that uses this field is installed and runs on an OPPO smartphone with SDK version 26, compatibility issues may arise.

Override/Callback: Due to the evolution of the Android framework, some callbacks may have been altered. Here, the callbacks are methods defined by the framework that could be explicitly overridden[4] by client Android developers, and their execution will be triggered by the framework. Figure 5.8 demonstrates such an example excerpted from [66]. The *onAttach(Context)* callback method at Line 1 is introduced from API level 23. This callback method will not be executed if this code is run on a smartphone with an API level lower than 23. Thus it could cause the *mActivity* field not to be initialized, and a "NullPointerException" may be thrown when using it.

The table shows that most of the tools are developed for detecting compatibility issues induced by the method evolution of the Android system. For the field evolution-induced compatibility issues, ACID and IctApiFinder have mentioned the issue in the corresponding papers but did not explain the issue in detail. Pivot and FicFinder also considered compatibility issues induced by methods provided by specific devices, while none of the detection tools examined compatibility issues resulted from fields carried by specific devices. For the independent issue induced by the evolution of callback methods, CIDER is the only approach developed intentionally to handle this, while ACID considered both evolution-induced and this special one. To summarize, unfortunately, none of these approaches have considered all the identified types of compatibility issues. The most recent approach, ACID [24], can only handle three out of the aforementioned five types, leaving device-specific issues unaddressed. It is also worth noting that the two approaches, which have indeed taken evolution-based fields into account, have only

[4] Actually, all the methods that are declared as public or protected could eventually be explicitly overridden by client apps. In this work, we take all of such methods into account and hence will not differentiate (and hence specifically emphasize) if the given method is a callback.

mentioned this capability but do not elaborate further with the support of experimental evidence.

Furthermore, in column 7 of Table 5.4, we further summarize whether the proposed approach involves a systematic approach to harvest an incompatible API list (hence the results can be considered complete). As summarized in Table 5.4, only three approaches (i.e., ACID, CiD, IctApiFinder) leverage a systematic approach to harvest incompatible APIs. The majority of considered approaches only take ad-hoc approaches aiming at detecting as many compatibility issues as possible without endeavoring to identify all the possible compatibility issues, i.e., the compatibility issues are not discovered following a systematic approach aiming at covering all the possible cases. As an example, Scalabrino et al. [20] present an automated compatibility issue detection approach called ACRYL, which leverages the knowledge collected from changes implemented in other apps responding to API changes to achieve its purpose. Such an approach, although implemented in an automated manner, cannot collect all the possible compatibility issues lying in the Android ecosystem and thereby can unfortunately yield false-negative results.

Finally, the last column further highlights whether the proposed approach itself is fully automated or not. An automated approach should not involve any manual efforts that may pose difficulties to replicate. Among the selected nine approaches, six of them do provide automated ways to identify compatibility issues (i.e., misuse of incompatible APIs) in real-world Android apps. Three approaches rely on manual efforts to achieve their objectives, making them not extensible (at least in an easy way) to detect newly introduced compatibility issues. For example, Wei et al. [19, 138] have empirically studied the fragmentation-induced issues to portray the symptoms and root causes of compatibility issues and subsequently proposed a static-analysis tool named FicFinder to detect such compatibility issues. The major limitation of FicFinder is the requirement of manual efforts to build the patterns of API/context pairs, which are summarized from the aforementioned empirical study. Such manual efforts are expensive to be extended to summarize more compatibility issues.

RQ1 Findings

In our analysis, no state-of-the-art approaches are capable of detecting all five types of compatibility issues that have been identified to date, and some of them require considerable manual effort.

5.3 Replicability Study (RQ2)

The second research question aims at checking to what extent can we replicate the experimental results yielded by the state-of-the-art tools targeting compatibility issues detection.

5.3.1 Tool Selection

Ideally, we would like to consider all the tools to perform the replicability study. Among the nine primary studies, there are, in total, seven tools worth reproducing. ACRYL and FicFinder have respectively been first presented in a conference paper and then extended to a journal paper. In these two cases, only the tool versions presented in the latest paper are considered. Among the seven tools, we decide to exclude Pivot as it does not really involve the actual detection of compatibility issues in Android apps, as highlighted in Table 5.3. For the remaining six detection tools, we download all of these different tools from their published site and contact the authors of the tools to make sure if the tools *per se* and the experimental datasets are the same as they were presented in the original papers. The developers confirm that IctApiFinder [10] has been updated due to the evolution of dependencies. We then try to execute them one-by-one in our local environment to make sure they can be successfully reproduced. Unfortunately, we have to further exclude ACID and ACRYL from consideration as these two tools cannot be successfully executed. We have contacted their authors for clarification, but until now, we still cannot properly execute them. Therefore, we conduct the reproducibility study based on the remaining four tools, which are detailed as follows.

CiD [14] first models the lifecycle of publicly available Android APIs by extracting Android APIs from Android framework source code and then analyses Android Apps including both the primary app code and extra code. However, it is uncertain whether the

Android app has accessed a problematic Android API or not just by checking if the app contains an invocation of the problematic Android API as the problematic Android API can also be protected by SDK version checkers. Therefore, the authors proposed a path-sensitive inter-procedural backward data-flow analysis to verify if the problematic Android APIs are protected with API-level related conditions. A compatibility issue is identified once the API is not protected by version condition checks and the API is not supported in the range designated in AndroidManifest.xml.

IctApiFinder [10] first conducts an extensive empirical study over 11 consecutive Android versions and approximately 5,000 Android Apps. The authors find that many different APIs are released between two consecutive Android API releases and thus App developers or third-party library developers provide additional code to guarantee the same behaviours on different OS versions. More importantly, the additional supporting code shares the same pattern that is SDK version check. With the provided SDK version check, different Android APIs are invoked to run smoothly on different OS versions. Based on these findings, they propose the tool by first building the interprocedural control flow graph (ICFG) by Soot for Android Apps and then extracting Android APIs (including the publicly available Android APIs and the restricted APIs with the access modifier protected) from SDK (android.jar) file as the authors believe that it is not accurate to extract from the SDK document api-version.xml. With the ICFG, it transfers the dataflow analysis problem into a reachability problem. For each Android API in the ICFG, the tool detects if it is supported in the defined API levels interval in AndroidManifest.xml as there are different SDK version constraints (conditional SDK version check to access the Android APIs) in different program points. If the designated API levels are not supported at a certain point, an issue is detected.

CIDER [66] focuses on compatibility issues caused by callback APIs as the evolution of Android frameworks. With the help of an empirical study, they find that two common types of callback API evolutions: API reachability change and API behavior modification can change app control flows and induce compatibility issues. Thus, they leverage the concept of Callback Control Flow Graph (CCFG) [141] and propose a graph-based model, Callback Invocation Protocol Inconsistency Graph (PI-Graph), to capture the structural invocation protocol inconsistencies to detect callback-induced compatibility issues (inconsistent app control flows) when apps running on different API levels. The

authors first encode seven different PI-Graphs related to 24 key Android APIs from their empirical dataset and then implement the detection tool based on Soot [142].

FicFinder [138] is actually the first seminal work to better understand fragmentation-induced compatibility issues and detect these issues via the proposed approach. By conducting empirical study and investigating real-world compatibility issues, the authors found that the majority of the issues are induced by the improper use of Android APIs in a problematic running environment, which is called issue-triggering context and the context can be expressed in context-free grammar. Therefore, the algorithm identifies the issue-inducing Android APIs as well as their dependencies, analyses the calling context, and then compares it with the modeled issue-triggering context. To analyze the dependencies issue-inducing API related, the algorithm carries out an inter-procedural backward slicing on the call site to acquire the slices of statements on the basis of program dependence graph [143]. If the triggering context is not considered before invoking the API, a new issue is reported. To implement this artifact, the well-known static analysis framework, Soot [142], is utilized.

Each of the selected tools requires a specific configuration. As the detection result relies on these basic configuration parameters, we investigated the tool document and configuration setup process and tried to align the configurations between these selected tools to make sure they do have a similar configuration.

5.3.2 Datasets

Recall that, with RQ2, we are interested in evaluating the replicability of the selected tools. We aim to achieve this by running the tools against their original datasets. We, therefore, request the tools' authors to share their datasets, including mainly the ones with results manually confirmed by the authors[5] and have been explicitly discussed in their manuscripts (hence can be compared). To this end, we have eventually selected 65 apps, which are made up of (1) twenty Android apps for CIDER, seven apps for CiD, eight apps for IctApiFinder, and thirty for FicFinder.[6]It is worth reminding the readers

[5]We decide to not request the full dataset leveraged by the authors because it may involve a very large number of apps that are not convenient to share.

[6]The FicFinder authors have actually considered 53 Android projects but only thirty of them can be compiled into Android APKs. Although FicFinder can take either Android APKs or disassembled class files as input, we will only replicate the capability of analyzing Android APKs, which are also the input of the other considered tools.

that we have to exclude some of the shared apps because they are no longer available on the web and hence the apps cannot be downloaded based on the information shared by the authors, or the shared source code snippets cannot be compiled to Android apps. Nevertheless, this exclusion of a small number of apps including 12 from IctAPIFinder and 23 from FicFinder $(12 + 23 = 35)$ should not impact the results of the replicability study.

5.3.3 Results

When carried out our replication, CIDER and CiD were found to have exactly the same outputs on the original Android Apps. In contrast, FicFinder and IctApiFinder have some different outputs with regard to their original experimental Apps. We now detail their differences respectively.

IctApiFinder. The artifact was developed along with the paper in 2018 and was not open-sourced till 2021. With the acquired eight exact Android Apps, we can successfully run the tool on all of them. However, 6 of them do have a different number of issues reported compared with the original paper. Among the six different apps, the paper in total reported 49 issues regardless of TP (True Positive) and FP (False Positive), while our experiment reveals 108 compatibility issues. As we cannot obtain the original results rather than the reported number of issues, we cannot know which issues are different compared with the original results. One reason explaining the differences could be that, as also confirmed by the authors, the tool has not been maintained during the last three years. Therefore, there are some dependencies that are not available anymore, and also, there are some APIs not supported in the newer updated dependencies. To release the project, the authors replaced it with newer versions of dependencies and commented out some non-supported APIs in the project. The authors further noted that they could not make sure if such updates have bad or good effects on the final detection results.

FicFinder. The artifact was first published in 2016 and then extended in 2018. We can successfully execute the artifact on all of the Android projects. The paper describes the detected results in two different categories. One is compatibility issues in TP and FP, and the other is Good Practice (GP) meaning already fixed issues. After we reproduce the artifact with the original experimental Android app dataset in our local environment,

seven of them have different outputs compared with the original ones presented in the original published paper. Among the seven apps, we find that 2 of them have the same total number of detected results but have a different number of compatibility issues and good practices, such as GadgeBridge [144] was reported one detected issue (regardless of TP and FP) and one GP but we reproduced with 2 issues detected, AnkiDroid [145] was reported 4 GP detected but we reproduced with 4 issues. The remaining five apps further have a different total number of detected results, such as LibreTorrent [146] revealed 6 GP but we detected only 3 GP, MozStambler [147] contained 1 issue and one GP but we only detected 1 issue. The possible reason behind this is that they did some regular updates on the artifact as the authors still utilize this one in their research, such as the case study in their newer work Pivot.

To summarize, as revealed by our study, most of the experimental results yielded by the selected four tools could be reproduced. The small number of cases that cannot be reproduced are mainly due to tools' updates, either because of lacking maintenance so that we have to arbitrarily update some dependencies to make it runnable in practice or intentional evolutions to keep improving its capabilities. Such updates, either intended or not, have indeed caused difficulties in reproducing the exact original results. Therefore, we argue that there is a need to always record the artifacts, along with the experimental datasets such as Android apps including both source code and bytecode APK files if possible, in permanent sites (e.g., Zenodo or Figshare). The artifacts should also be well-configured in docker-alike containers that can support direct execution of the tools and hence mitigate unnecessary dependency errors that may hinder the tools' replicability.

RQ2 Findings

Most of the experimental results yielded by the four selected state-of-the-art tools can indeed be reproduced. There are, however, a small number of non-replicated cases that are mainly caused by slight updates of the tools or the evaluated apps.

5.4 Comparison Study for Issue Detection (RQ3)

This research question aims to empirically compare the state-of-the-art tools targeting the detection of compatibility issues in Android apps. We rely on the validity provided by

the artifacts themselves. We answer this research question by first presenting the experimental setup (including tool selection and datasets) in Subsections 5.4.1 and 5.4.2 and then the experimental results in Subsection 5.4.3.

5.4.1 Tools Selection

Recall that there are only four tools that we can replicate to scan compatibility issues (as discussed in the previous section). Therefore, we select the same four tools to achieve this objective in this work, i.e., comparing these four tools w.r.t. their compatibility issues detection capabilities.

5.4.2 Datasets

In this work, we use the following two datasets to support our comparison study in compatibility issues detection.

- **Dataset1**: The same 65 apps used for the replicability study as discussed in Section 5.3.
- **Dataset2**: 645 Android apps selected from the total 1,375 open-source apps of AndroidCompass dataset [123]. AndroidCompass contains a dataset of git commits related to Android compatibility checks (including evolution-induced, device-specific, and override/callback-related ones), which are originally harvested from 1,375 open-source Android projects on Github. Some git commits contain compatibility issue fixes (e.g., adding compatibility checks for APIs that are not protected initially), while others do not (e.g., adding new Java files that include compatibility checks). In this work, we are only interested in the former ones as based on which we could locate problematic app versions containing actual compatibility issues. This study will leverage this information as partial ground truth to support the comparison study. We collect Android projects with git commits containing API version guard checks (i.e., checking the Android versions just before the API invocations). Going one step further, we reset the commit of the selected projects just before the fix commit (i.e., adding Android version checks) and compiled them into installable APKs. Unfortunately, several app projects are no longer available

TABLE 5.5: Experimental results obtained based on the 65 apps located in Dataset1.

App Name	Callback-induced	Evolution-Induced		
	CIDER	FicFinder	IctApiFinder	CiD
Tinfoil-Facebook	0	1	1	2
kolabnotes	1	4	5	6
SteamGifts-chocolate-debug	0	6	20	23
OsmAnd	1	3	7	44
iFixitAndroid	0	7	58	218
Simple-Solitaire	1	0	1	10
Anki-Android	0	6	150	86
login-sample-debug	4	0	2	3
ooniprobe-android-1.3.1-debug	1	0	4	38
APICompatibility_Inheritance	0	0	2	3
APICompatibility_Varargs	0	0	2	2
SurvivalManual-4.1-debug	0	2	1	15
Calendula	0	15	29	63
libretorrent	3	1	13	59
APICompatibility_Protection2	0	1	1	0
StreetComplete	0	2	7	5
red-moon	0	0	13	21
padland	1	0	13	4
duckduckgo-0.6.0-release	1	0	1	2
transdroid	0	1	214	37
materialistic-hacker-news	0	1	32	36
materialfbook	0	1	15	38
ownCloud	0	2	66	181
AndStatus	0	2	43	27
RedReader	0	1	30	7
opentasks-1.1.8.2	0	24	14	51
APICompatibility_Basic	0	0	1	1
Gadgetbridge	0	2	21	35
Total	13	82	766	1,017

on Github, while some others cannot be easily compiled into APKs (e.g., due to outdated dependencies), so we have to exclude them. Eventually, we were able to collect 645 apps (at least one compatibility issue in each of them) to build this dataset.

5.4.3 Results

Results on Dataset1. We first apply the selected tools to analyze the apps in Dataset1. Unfortunately, 37 apps cannot be handled successfully by both IctApiFinder and CiD (i.e., 24 and 15 failures, respectively). The corresponding error messages, such as worker thread execution failed [7], Dex version is not supported [8], and IllegalArugmentException [9], etc., indicate that the failures are mainly raised by Soot, the underlying static

[7] https://github.com/soot-oss/soot/issues/1279
[8] https://stackoverflow.com/questions/49606951/dexexception-not-support-version
[9] https://github.com/soot-oss/soot/issues/331

analysis framework leveraged by these two tools. This problem has been discussed by the authors in their article as a potential threat to validity. It is also a well-known problem when performing static analysis on top of Soot.

For the remaining 28 successfully analyzed apps, Table 5.5 presents the detection results. CIDER, different from the other three detection tools, was developed for callback-induced compatibility issues. Among the 28 apps, only 8 apps are reported to include callback-induced issues. The reason behind this small number could be explained by the fact that the tool only leverages seven manually summarized rules to detect such issues. Such a manual process may not be able to include all the different situations and hence may lead to incomplete results. Similarly, FicFinder, which leverages 20 manually summarized incompatible APIs, reports only 82 compatibility issues, which are also significantly fewer results compared with the remaining two tools that have leveraged systematic approaches to harvest incompatible APIs (as indicated in Table 5.3). The typical working process for these selected issue detection tools as shown in Figure 5.3 in the paper includes two steps. The first one is incompatible APIs gathering and the second step is issue detection via program analysis by incorporating the incompatible APIs harvested in the first step. The number of collected incompatible APIs is strongly correlated with the final detected number of compatibility issues. The number of incompatible APIs leveraged by CiD and IctAPIFinder is much more than others. The more incompatible APIs imply more potential incompatibility issues. This experimental result further confirms that it is essential to invent systematic approaches to harvest incompatible APIs so as to support automated compatibility issues detection in Android apps.

Both IctApiFinder and CiD yield significantly more results than FicFinder (revealed by the total number of detected issues in Table 5.5 classified into two types, including Callback-induced and Evolution-induced, according to the targets of the selected detection tools) since they do take systematic approaches to collect much more incompatible APIs compared with the limited manually harvested ones in FicFinder, which demonstrates the importance of the incompatible APIs gathering for the issue detection. However, the detection results of IctApiFinder and CiD are quite different. Among the 28 apps successfully analyzed by both of these two tools, IctApiFinder and CiD respectively yield a total of 766 and 1,017 issues, for which only 52 were reported by both of them. This experimental result is quite surprising as we would have expected

that IctApiFinder and CiD would have much more overlap in terms of their detected compatibility issues. We, therefore, go one step deeper to investigate why these two tools yield quite different results, i.e., being able to locate a quite number of compatibility issues while also missing many of them reported by the other tool. We look at the number of distinct incompatible APIs detected by these two tools. Our analysis shows that the total 766 and 1,017 compatibility issues reported by IctApiFinder and CiD are essentially caused by 147 and 551 incompatible APIs, respectively. As highlighted in Figure 5.10, the intersection between these two incompatible APIs sets is quite small (i.e., only 63 out of 551 incompatible APIs considered by CiD are also taken into account by IctApiFinder). One reason causing this difference is that different framework versions are considered (e.g., the incompatible APIs collected by IctApiFinder are from 4 to 27, while CiD is from API 1 to 25). Subsequently, the common compatibility issues reported by both of these two tools will be small as well. We could not configure the same incompatible APIs from the same range of API levels as different detection tools require different inputs. For example, FicFinder requires JSON-like configuration of incompatible APIs as input and IctApiFinder takes additional third-party tools, such as Heros [62], Doop [63], LogicBlock [64, 65], etc., to extract incompatible APIs. Even though a fair comparison is not possible, we could still have such a comparison to have a basic understanding of the ability of issue detection among the collected detection tools. As is known to us, the attribute `minSDKVersion` indicating the minimum API level on which the apps could be run is always greater than API level 4 among nowadays available apps. Among the dataset of a total of 710 $(65 + 645)$ Android apps in our experiments, only 4 of them set the `minSDKVersion` to 4, only accounting for 0.56% (4/710). The lower API level of the API range does have little impact on the issue detection results.

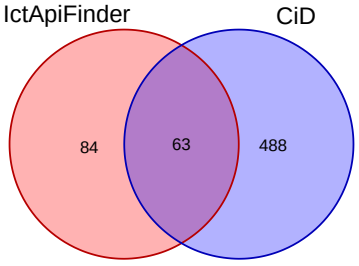


FIGURE 5.10: Venn diagram of incompatible APIs utilized in IctApiFinder and CiD.

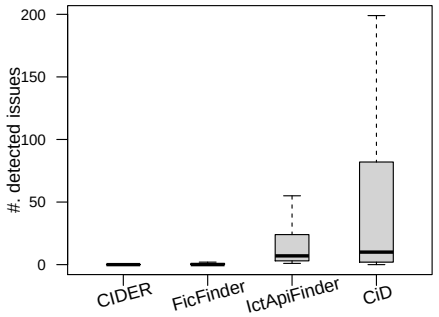


FIGURE 5.11: Compatibility issues detected by different detection tools against Dataset2.

Results on Dataset2.

We then apply the selected tools on Dataset2, which contains a large number of real-world Android apps selected from the AndroidCompass compatibility checks dataset. Unfortunately, over half the apps are excluded from the dataset as they cannot be successfully analyzed by all the selected tools. Among the 277 remaining apps, CIDER, FicFinder, IctApiFinder, and CID have reported 12, 277, 5,009, and 27,874 compatibility issues, respectively. Figure 5.11 further illustrates the distribution of detected compatibility issues in real-world Android Apps. Clearly, CiD yields more issues than the other tools, followed by IctApiFinder and then FicFinder. CIDER reports the least number of compatibility issues. These differences are also significant as confirmed by a Mann-Whitney-Wilcoxon (MWW) test at a significant level[10] at 0.001.

We note that this experiment, although with a large number of apps, supports the same findings discussed previously. First, there is a strong need to invent systematic approaches to harvest compatibility issues detection rules (i.e., identifying incompatible APIs). As shown in Figure 5.11, the number of issues reported by CIDER and FicFinder (with manually summarized rules) is significantly less than that achieved by IctApiFinder and CiD (with systematically harvested rules). Furthermore, the fact that the intersection between the results yielded by the selected tools is quite small suggests that existing tools could be leveraged to complement each other. This result further shows that there is still a gap in the community to implement promising approaches to flag compatibility issues in Android apps, i.e., the capability of detecting compatibility issues has not been mature. Last but not the least, we believe that it is not exactly fair to directly compare existing tools targeting compatibility issues detection in Android apps as the evolution of the Android ecosystem is very fast. Tools developed at different times will likely collect a different set of incompatible APIs (e.g., the incompatible APIs collected by IctApiFinder are from 4 to 27, while CiD is from API 1 to 25), which subsequently will lead to a different set of compatibility issues. Therefore, we argue that, when comparing compatibility issues detection tools, there is a strong need to make sure that the underlying set of incompatible APIs is kept the same, which is however non-trivial to achieve as existing tools may not always be made open-source.

[10]Given a significance level α = 0.001, if p-value ¡ α, there is one chance in a thousand that the difference between the datasets is due to a coincidence.

RQ3 Findings

CiD is able to yield more compatibility issues than the other tools. However, their results do not overlap much, suggesting that existing tools are complementary to each other and yet still have limitations to achieve comprehensive compatibility issues detection. Furthermore, the fact that CiD and IctApiFinder can yield significantly more results than FicFinder and CIDER suggests that it is essential to leverage systematic approaches to mine incompatible APIs so as to support the comprehensive detection of compatibility issues.

5.5 *AndroMevol* and Its Effectiveness Evaluation

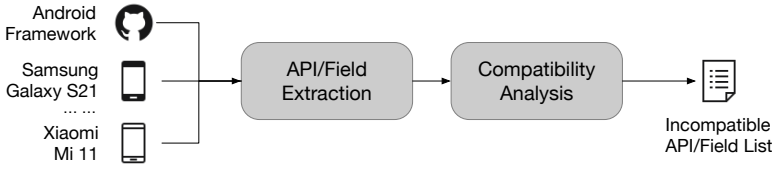


FIGURE 5.12: The working process of *AndroMevol*.

5.5.1 Approach

The working process is illustrated in Figure 5.12. *AndroMevol* is responsible for experimentally pinpointing incompatible APIs, which could be accessed by Android apps that subsequently will suffer from compatibility issues. While existing approaches are either labor intensive or only the evolution of the official Android releases is considered, *AndroMevol* not only takes the evolution of official Android releases but also the evolution of third-party Android customizations into consideration. It takes as input a set of Android frameworks extracted from various sources (i.e., the official Android open-source code and customized ROMs released by different manufacturers) and outputs a list of incompatible Android APIs through occurrence detection. It achieves this through two key steps, (1) API Extraction and (2) Compatibility Analysis. We now detail these two steps, respectively.

5.5.1.1 Step 1 – API Extraction

We want to systematically identify all the incompatible Android APIs, including both device-specific APIs and evolution-induced ones. Concerning APIs, we extract methods and fields modified by the access modifier public or protected both from java classes and interfaces. Additionally, the examined APIs not only include the publicly provided ones [148] but also those restricted APIs (i.e., non-SDK APIs) [16, 31]. It is worth mentioning that we do not take the implementation of the collected APIs into account as we cannot extract the implementation of the APIs easily and we either cannot determine if the implementation update of the APIs would eventually induce compatibility issues. We will discuss the effects of such extraction on threats to validity. To this end, we need to consider as many versions of Android frameworks as possible, including the official ones mainly maintained by Google and the customized ones provided by different Android manufacturers such as Samsung and Xiaomi (i.e., hereinafter referred to as brands). For each brand, we need to consider one framework at each Android API level.

In practice, this is non-trivial to achieve. First, it is expensive to purchase a complete set of physical devices covering all the brands. Yet, some brands (or specific versions of those brands) are no longer available on the market and hence cannot be purchased. Therefore, for simplicity, we will only consider the major brands and resort to their publicly released ROMs (instead of the physical phones) to locate the bytecode of Android frameworks. To determine the Android version of the released ROMs, we extract the ROMs and check the attribute of `ro.build.version.release` in the file `build.prop`. Second, during the evolution of the Android system, the framework code has been relocated to different locations or even changed into other formats that are hard to interpret.

To overcome this, we propose a best-effort approach to disclose incompatible Android APIs. The more frameworks provided, the more complete results the approach will yield. For the official Android frameworks, to ensure full coverage (one framework per API level), we resort to the official Android Open Source Project (instead of the released ROMs) to locate Android APIs. To support the analysis of both Android framework source codebase and bytecode versions (often named as *framework.jar*), we implement in this module two parsers.

Source Code Parser. This parser directly parses all the Java files located in the Android framework codebase project.[11] For each Java file, it records all its publicly defined methods and fields. Inspired by the approach proposed by Li et al. [83], we further refine the previous results by taking into account the following features:

Inheritance: A sub-class will implicitly extend all of its super classes' public or protected methods even if it does not explicitly redefine them. Similarly, when a given redefined method is dropped out in the sub-class, there is still a need to keep track of it as it could still be available (e.g., implicitly inheriting from its superclass, for which it has not yet been dropped out). These situations need to be carefully taken into account in order to achieve accurate results.

Generic type: Generic type is a Java feature that parameterizes types for convenience. An example of generic type could be the second parameter (i.e., E) defined in method *¡LinkedList: E set(int, E)¿*. This parameter (or generic type) will make it complicated to syntactically match its usages in practice, e.g., its usages could be both *set(int, String)* and *set(int, Float)*. This feature needs also to be resolved.

Varargs: Varargs is a Java feature that challenges the syntactic detection of API usage in Android apps. Varargs (defined as *TYPE...*) allows methods to receive an arbitrary number of parameters, which traditionally can be done through an array. For example, the *MessageFormat.format* method is declared as *format(String, Object...)*. The three dots after the final parameter's type indicate that the method can receive two or more parameters in practice, e.g., the actual usage of this method could be *format(String, Object)* or *format(String, Object, Object)*, etc.

Bytecode Parser. The implementation of this parser is more straightforward. Similar to the source code parser, it leverages Soot to go through all the Java classes in the jar file and record all the publicly defined methods and fields. It then goes one step deeper to take *class inheritance* into consideration to refine the results. The *generic type* and *varargs* features are ignored in this parser as they are expressed in different formats compared with the expressions in source code (i.e., the expressions such as ⟨*LinkedList: E set(int, E)*⟩ and *format(String, Object...)* do not exist in the decompiled Java files).

[11] https://github.com/aosp-mirror/platform_frameworks_base

During our analysis, we noticed that the APIs extracted from the source code repository and the framework.jar bytecode have some differences, i.e., some methods in the source code repository are not included in the framework.jar, and vice versa. By comparing the source code repository with its corresponding official framework.jar (at the same API level), we realize that some packages exist only in the source code but are not packaged into the framework.jar. Similarly, many APIs extracted from the bytecode do not exist in the source code. That's because the same source code files in different releases or different customizations are packed into different package files. This difference will unavoidably impact the subsequent compatibility analysis. To mitigate such an impact, we resort to a filter to exclude packages that only exist in source code or bytecode. The filter eventually contains 3,148 package names, which have been subsequently applied to all the API extractions, no matter which parser is used.

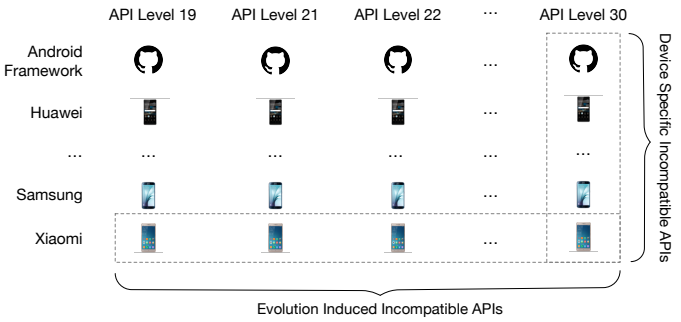


FIGURE 5.13: Incompatible APIs identification in Compatibility Analysis.

```
1  public <U> void method1(U t1)
   {
2    ...
3
4  public <E extends java.lang.
   String> void method2(E t2) {
5    ...
6
7  public void method3(java.lang.
   String... t3) {
8    ...
9
```

FIGURE 5.14: Java source code with generic types and varargs

```
1  public void method1(java.lang.
   Object) {
2    ...
3
4  public void method2(java.lang.
   String) {
5    ...
6
7  public transient void method3(
   java.lang.String[]) {
8    ...
9
```

FIGURE 5.15: Jimple code generated with Soot

FIGURE 5.16: Code example for generic types and varargs.

5.5.1.2 Step 2 – Compatibility Analysis

The second module of *AndroMevol* takes the extracted Android APIs as input and pinpoints potential incompatible methods and fields that could cause runtime crashes or unexpected behaviors if accessed by Android apps without appropriate guard checks. It aims at generating a complete list of incompatible APIs that could be leveraged to support existing or emerging app analysis approaches to soundly pinpoint practical compatibility issues in Android apps.

The approach we take to identify incompatible APIs is represented in Figure 5.13. Given a method or field in a framework (either official or customized), we consider it as an incompatible API introducing compatibility issues if (1) within the same brand, it exists in some versions but not in others (the evolution induced incompatible APIs as is shown in every single row in Figure 5.13) or (2) at the same API level, it exists in the frameworks of some brands but not in others (the device-specific incompatible APIs as is shown in every column in Figure 5.13). Following this rule, this module visits every method and field in each framework to locate potential incompatible APIs, which are then put into a single configuration file for easing external usage.

Because *generic type* and *varargs* features are not involved in the bytecode parser, the results yielded by the bytecode parser will be slightly inconsistent with the ones yielded by the source code parser as we collect APIs both from source code and the file framework.jar in released ROMs. To mitigate this issue, we put additional effort into this module to unify the results before conducting the actual compatibility analysis. Figure 5.16 shows an example of how these features are represented in intermediate representation for compiled Java programs. Figure 5.14 represents the declaration of methods with generic types and varargs as parameters while Figure 5.15 displays the corresponding Jimple format, which is the principle Intermediate Representation and basic analysis unit in Soot [149]. Finally, the unification is done by applying the following rules to the results yielded by the source code parser.

method(T) $\rightarrow$ *method(java.lang.Object)*

method(T extends java.lang.Util) $\rightarrow$ *method(java.lang.Util)*

method(java.lang.String...) $\rightarrow$ *method(java.lang.String[])*

TABLE 5.6: Distribution of exclusive/absent methods and fields in different API levels at different vendors. Recall that for the official platform, we have recorded the evolution-specific results for all the available Android API levels. This table only presents the results from version 19 to 30 for comparison purposes.

API level	API	Official		Huawei		Xiaomi		Oneplus		OPPO		Samsung		Total (Distinct)
		Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	(Methods/Fields)
19	Methods	765	21,498	1,428	16,148	324	20,478	42	17,105	1,831	15,844	12,606	5,206	22,938
	Fields	88	28,533	4,864	20,245	211	26,538	40	24,671	6,665	18,503	12,540	12,524	28,670
21	Methods	579	24,347	3,531	18,916	4,472	17,561	769	21,294	33	21,938	12,434	9,915	24,955
	Fields	25	28,140	5,758	19,560	5,569	19,280	650	24,235	18	24,815	12,693	12,372	28,165
22	Methods	585	36,669	4,010	30,604	9,071	25,083	704	33,369	5,922	28,801	13,491	21,051	37,290
	Fields	25	45,240	6,542	35,677	8,418	33,122	1,812	39,946	10,434	31,530	13,947	27,985	45,281
23	Methods	580	32,134	2,139	27,976	8,424	21,299	532	29,185	3,356	26,351	14,544	15,344	32,723
	Fields	25	39,386	3,723	32,458	6,279	29,195	2,804	32,660	6,540	28,891	15,929	19,914	39,413
24	Methods	1,518	31,851	2,777	27,478	7,949	22,100	-		233	30,074	17,445	12,957	33,371
	Fields	408	36,930	5,175	27,941	7,486	25,260			1,021	32,766	18,565	14,609	37,340
25	Methods	1,534	34,705	-		7,852	24,916	585	32,245	4,700	28,371	18,037	15,251	36,246
	Fields	415	40,324			7,538	28,399	809	35,172	8,135	27,961	18,944	17,538	40,741
26	Methods	1,155	49,485	3,978	33,442	8,343	28,835	687	36,702	4,755	48,514	18,056	19,730	55,464
	Fields	372	53,367	9,420	32,956	6,898	35,040	1,085	40,958	4,852	49,152	19,064	23,501	56,769
27	Methods	1,064	41,635	3,405	35,633	8,709	30,212	115	39,055	5,910	32,970	19,462	19,821	42,706
	Fields	347	74,642	31,195	37,716	6,594	61,938	72	68,952	9,869	58,603	20,194	48,874	74,989
28	Methods	700	61,900	4,993	52,667	15	57,390	19,471	35,732	7,343	49,928	19,843	37,828	63,629
	Fields	61	63,080	12,073	41,844	23	53,302	7,589	47,565	12,533	40,581	19,148	34,732	63,634
29	Methods	392	70,555	6,905	50,192	11,621	45,191	1,743	54,899	9,048	47,627	26,812	30,334	70,962
	Fields	36	66,569	12,052	46,243	8,556	49,232	2,051	55,630	13,965	43,715	20,934	37,321	66,605
30	Methods	443	79,750	-		13,455	43,557	2,526	54,401	11,989	57,321	28,494	28,899	111,064
	Fields	110	70,590			9,240	47,223	9,072	47,370	15,332	48,205	22,605	34,405	72,040
Total (Distinct, Device-specific)		3,473	346,673	59,805	214,247	32,548	273,885	42,456	260,937	52,939	300,003	108,082	196,486	397,678
Total (Distinct, Evolution-induced)		Exclusive: 7,374 Absent: 23,126												

The rules are built based on our observations on how *generic type* and *varargs* features are handled by the Java compiler and Soot. After applying these rules, the *generic type* and *varargs* no longer persist in the extracted Android APIs, resulting in unified results that also simplify their potential usages (i.e., no need to consider *generic type* and *varargs* features anymore).

5.5.2 Effectiveness of *AndroMevol* (RQ4)

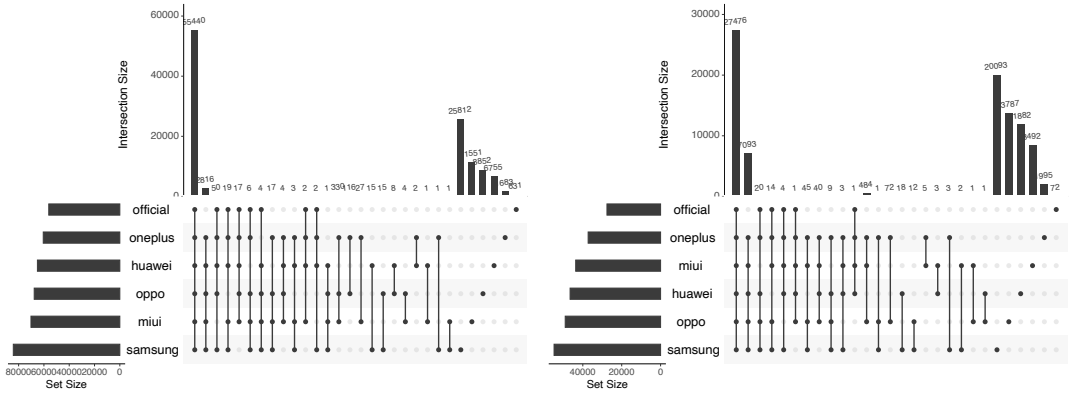


FIGURE 5.17: UpSet plots of the intersections of methods (left) and fields (right) offered by different vendors at API level 29. An UpSet plot is made up of three parts: (1) The left barplot presents the total size of each set, (2) The bottom plot presents all the possible intersections, and (3) The top barplot presents the occurrence of each intersection marked in the bottom plot.

This research question concerns the effectiveness of our *AndroMevol* prototype itself in incompatible API gathering. We set out to evaluate if it is capable of automatically

generating a list of incompatible Android APIs (i.e., methods and fields) for helping the community spot potential compatibility issues in Android apps.

5.5.2.1 Datasets

For the sake of simplicity, in addition to the official Android system, we select five additional platform vendors for this experiment. The five selected vendors are Samsung, Xiaomi, Huawei, OPPO, and Oneplus. We chose these because they are among the most popular Android brands in the market. For the official Android system, we consider all its platform versions as they are all recorded in the open-source repository. For the remaining five vendors, we consider one platform version at each API level, ranging from 19 (i.e., the minimal version that still holds over one percent of distribution) to 30 (i.e., the latest version at the time when we conducted this study). Ideally, we should additionally consider 55 (5 platform vendors * 11 platform versions) different Android platform versions. Unfortunately, it is non-trivial to collect those frameworks related to specific vendors. Indeed, the vendors have neither maintained a complete release document nor recorded the AOSP version in their released ROMs. As a result, we have to download ROMs from different sites (notable ones include [150–152] etc.) to locate the correct ones aligned with certain AOSP versions. This step has cost the authors more than a month to complete, and in total, we have downloaded and explored over one thousand ROMs. Still, we cannot successfully extract the platform frameworks for Huawei at API levels 25 and 30 and OnePlus at API level 24. To this end, we have to ignore those versions when applying *AndroMevol* to generate the list of incompatible Android APIs. Nevertheless, ignoring the three versions should not impact the reliability of the generated incompatible API list. That's because we do not take them into consideration when we determine if APIs are incompatible or not. Moreover, these missing ROMs hardly exist as we cannot find them in the wild (even if we have spent more than one month searching). Take Huawei as an example, its latest Android OS has been replaced by its HarmanyOS. The ROM with API level 30 customized from the official Android was not even published to the public.

5.5.2.2 Results

Table 5.6 summarizes our experimental results. **In total, *AndroMevol* identifies 397,678 incompatible APIs with both evolution-induced and device-specific APIs considered that do not exist in all the considered platform versions**. This list contains 388,819 device-specific ones that only exist in some vendors' platforms but not in others and 30,500 evolution-induced ones (i.e., summarized in the last row) that only exist in some framework versions but not in others even within the same vendor. It is worth noting that the same incompatible API could not only reside in device-specific APIs but also in evolution-induced ones. In total, 21,641 incompatible APIs both belong to device-specific and evolution-induced APIs and 23,459 distinct incompatible APIs were harvested. For example, an API is added in a newer official release. Some customized OSs merge such APIs but others may not, which would introduce both device-specific and evolution-induced APIs. Every platform contains APIs that are (1) exclusive to the platform version itself and (2) absent from the other platform versions at the same API level. This experimental result suggests that both the evolution of the Android framework and the customization of the official framework introduced by different Android vendors have introduced divergences among the different platform versions. Since a given Android app is expected to be installed and executed on all those platforms, such divergences may cause inconsistencies in offered Android APIs and, subsequently, cause app crashes on some platforms while being successful on others (i.e., compatibility issues). Additionally, it is worth noting that the number of incompatible APIs is increasing as the evolution of Android releases including both the official and customized ones, which would induce more compatibility issues if the developers do not handle such incompatible APIs properly during their development. To avoid such potential issues in released apps, developers should pay more special attention to their development while invoking such APIs. Moreover, OS maintainers could proactively reduce the number of incompatible APIs in their releases.

To evaluate the correctness of the harvested incompatible APIs, we resort to human efforts to check a set of sampled results manually. We first randomly selected 378

incompatible APIs from the selected detection tools, CiD, FicFinder, CIDER, and IctApiFinder as well as the incompatible API harvesting tool, Pivot [12], and then manually checked their incompatibility to set them as ground truth. With the validated ground truth, *AndroMevol* achieved a high precision of 82.24% and recall of 96.17% in identifying incompatible APIs. What's more, the incompatible APIs missed by *AndroMevol* are almost harvested from other packages, such as the external Unicode support [13], which is out of consideration of *AndroMevol* and we will try to resolve in future.

Figure 5.17 further illustrates with an example the incompatible methods/fields differences (via respectively two UpSet plots) between different platforms at API level 29. The left barplot in the UpSet plot represents the total number of elements from different sets. For example, the left UpSet plot in Figure 5.17 shows that Samsung provides the highest number of methods for developers while official gives out the lowest number of methods, which are represented in the left barplot with the number of methods over 80,000 and 60,000, respectively. The top barplot shows the detailed number of elements common or unique among different sets, such as the first bar with the number 55,440, which corresponds to the direct bottom dotted line and represents that the total number of 55,440 methods are common among all the six sets, and the last bar with number 631 representing that the official framework provides 631 unique methods comparing with other five vendors since there exists only one dot in the bottom plot. Clearly, and as expected, the majority of methods and fields are kept for all the considered platforms, while each of them has some exclusive ones. The fact that different combinations of intersections (for both methods and fields) exist **shows that there are indeed divergences among the selected platforms, which can lead to potential compatibility issues**. We observe that, perhaps surprisingly, **there are a significant number of methods and fields that are not available in the official framework but are available on all the other platforms.** Our in-depth investigation reveals that this phenomenon is caused by the fact that some APIs have been removed by the official framework during its evolution but are kept by customized frameworks. This suggests that **the customized framework developers do not always keep up with the updates of the official framework**. This confirms our previous finding [154] that there

[12] In total, we have 23,459 distinct incompatible APIs harvested and resort to the famous Sample Size Calculator [153] with a confidence level of 95% and a margin of error of 5%. It gave out a sample size of 378 based on the total sample size of 23,459. We, therefore, selected 378 incompatible APIs.

[13] https://developer.android.com/guide/topics/resources/internationalization

are divergences between the official Android framework and the selected customized frameworks.

RQ4 Finding

AndroMevol is effective in harvesting incompatible APIs. Among the official Android framework and its five branches, it has identified a total of 397,678 incompatible APIs, giving an accuracy of 98%.

5.5.3 Comparison with state-of-the-art tools in incompatible API gathering (RQ5)

The compatibility issue detection tools we selected for our replicability study all have their approach to harvesting incompatible APIs. It should be noted that these detection tools were developed first to gather incompatible APIs and then to feed such APIs to their analysis implementation to pinpoint compatibility issues. In addition, the identified compatibility issues are only induced by a subset of the collected incompatible APIs as we cannot enumerate all of these incompatible APIs in our collected finite app dataset. In contrast, our tool *AndroMevol* was designed to systematically identify incompatible APIs with both evolution-induced and device-specific ones considered. In our last research question to evaluate the performance of our approach, we propose to further compare the ability to identify incompatible APIs between our tool and the state-of-the-art detection tools – FicFinder, CIDER, CiD, and IctApiFinder. Our comparison study aims to address the following concerns: How many incompatible APIs are identified by these detection tools? Could the APIs pinpointed by our tool cover the incompatible APIs identified by other detection tools? If not, what is the reason for the differences in identifying the incompatible APIs?

Besides the detection tools we discussed in the previous comparative study, we re-involved the excluded API gathering-oriented tool, **Pivot** [23]. Pivot, different from the previous detection tools aiming to detect compatibility issues in apps with different approaches, is proposed to automatically learn Fragmentation-Induced API-correlations from released Android Apps. The tool first builds inter-procedural control flow graphs by associating the app's call graph and every method's control flow graph

and then traverses the whole graph to identify device-checking statements and evaluates the conditions on each branch. With the identified device-checking statements and the constraints on each branch, all reachable APIs with the device constraints are constructed as API-device correlations. At last, the paper resorts to two metrics: in-app confidence and occurrence diversity, to filter out invalid API-device correlations. The API-correlation is a tuple comprising of an API invocation and the guarded device-checking conditions, such as `<android.hardware.Camera$Parameters: void setRecordingHint(boolean)>, ‘‘Nexus 4’’>`.

5.5.3.1 Datasets

We extracted the incompatible APIs used in the selected four different detection tools, including CiD, IctApiFinder, CIDER, and FicFinder, and downloaded the incompatible APIs generated by the API harvesting tool Pivot. We involve the tool, Pivot, back because it was developed for incompatible APIs gathering from the published Android apps rather than issue detection, which we discussed in RQ3 and excluded it. We now detail the differences between the selected issue detection tools including CiD, IctApiFinder, CIDER, and FicFinder as well as Pivot and *AndroMevol* with regard to the ability in collecting incompatible APIs.

5.5.3.2 Results

Figure 5.18 represents the differences between the selected detection tools, including the newly considered API harvesting tool Pivot, and our proposed tool *AndroMevol*. In the UpSet plot, the left bar chart shows how many incompatible APIs were pinpointed by the detection tools. This bar chart clearly shows that our tool *AndroMevol* (with reporting 236,099 incompatible APIs) outperforms the other detection tools by pinpointing many more incompatible APIs.

The possible intersections represented in the top and bottom plots further reveal that most of the incompatible APIs identified by *AndroMevol* were actually missed by the other selected state-of-the-art detection tools. As illustrated in the first column of the

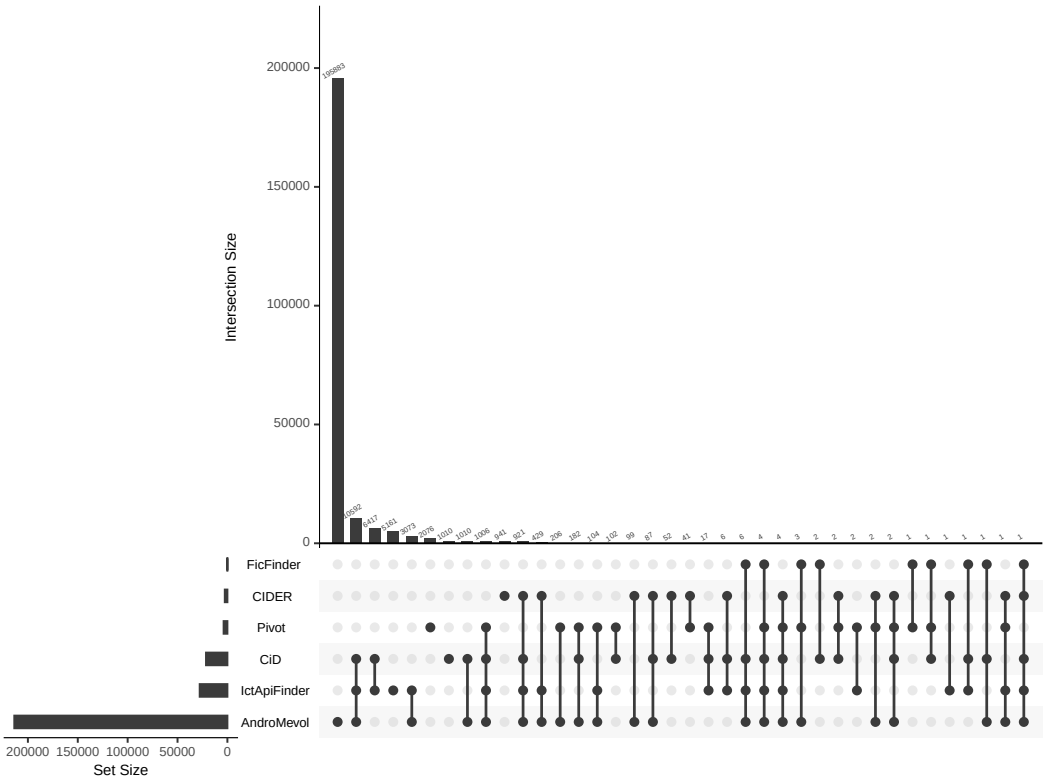


FIGURE 5.18: UpSet plot of incompatibility APIs among different detection tools. An UpSet plot is made up of three parts: (1) The left barplot presents the total size of each set, (2) The bottom plot presents all the possible intersections, and (3) The top barplot presents the occurrence of each intersection marked in the bottom plot.

vertical bar chart, 195,883 incompatible APIs are exclusively harvested by *AndroMevol*, accounting for 82.97% of the total number of harvested incompatible APIs. This result is expected as *AndroMevol* not only systematically records the historical changes of the official Android framework but also takes into account the various customizations done by five smartphone manufacturers. It is still worth mentioning that even though the majority of the collected incompatible APIs are unique to *AndroMevol*, *AndroMevol* still could cover a considerable number of incompatible APIs reported by our selected tools. For example, the number of incompatible APIs in the intersection among AndroMevol and CiD accounts for 82.70% (13,938/16,694) with regard to the incompatible APIs given by CiD.

Interestingly, even though *AndroMevol* significantly outperforms the other tools by harvesting much more incompatible APIs, it does miss some incompatible APIs within all sorts of types that are identified by the selected state-of-the-art tools. We, therefore, go one step further to manually check those incompatible APIs harvested and utilized

in the published tools and investigate why they cannot be identified by our tool *AndroMevol*. Our analysis shows the main reason is that the different approaches do have different Android eco-system coverage. *AndroMevol* mainly focuses on incompatible APIs with different signatures from the Android framework, but other tools except FicFinder take different Android packages and method semantics into consideration. We now discuss the identified differences in detail.

- FicFinder was the earliest detection tool to identify compatibility issues. It harvested incompatible APIs manually and contains only 20 incompatible APIs. The bottom plot also reveals that our tool *AndroMevol* could also cover most of the manually collected APIs but not all of them. To identify incompatible APIs, *AndroMevol* extracts incompatible APIs from *framework.jar* provided in Android ROMs, which mostly contains package of Android framework but not others. Since we do not have a consistent correspondence between Jar files in Android ROMs and their source code, it is non-trivial to extract APIs for other packages from other Jar files in Android ROMs. Therefore, we only took framework into consideration when we developed *AndroMevol*, which might miss some incompatible APIs provided in other Jar files.

- Pivot [23] was proposed to collect the API correlations by learning from real published Android apps to empower their compatibility issue detection tool (i.e., FicFinder [19]). Our in-depth analysis reveals that Pivot considers more than 1,000 different device-checking conditions, such as Vivo, Lenovo, Meizu, and HTC-related device checks, etc., which were not taken into account in our experiments (i.e., we only considered five brands at the moment). This result sheds light on our future work to consider more popular Android devices for harvesting an even more comprehensive set of incompatible APIs.

- CIDER focuses on compatibility issues triggered by invoking callback APIs. Some of the callback APIs are added during the evolution of Android, which can also be identified by *AndroMevol*. Some other incompatible APIs were detected because they have involved semantic changes (the signature has not been altered), which, by far, have been overlooked by our tool *AndroMevol* (cf., 5.6.2.1).

- CiD, the state-of-the-art compatibility issue detection tool, extracted incompatible APIs from the provided API summary in the AOSP source set. The summary contains APIs from not only the Android framework but also other packages. Unfortunately, the API summary is no longer provided for the newer Android releases. We, therefore, harvested incompatible APIs from the Android framework only (extracting APIs both from the source code of Android framework and *framework.jar* provided in Android ROMs) in our approach, resulting in some incompatible APIs missed by *AndroMevol* (i.e., only being reported by CiD). For example, APIs from class `android.net.wifi.p2p.WifiP2pDevice` are provided in the summaries but we cannot extract them from the source code of the Android framework. That's because the java file locates under the source set `packages` [14] rather `framework`. We focus on the Android framework only as it provides the basic services that Android apps directly and heavily rely on [12, 13, 25, 81–83, 96]. We do not extend the API levels support of CiD here because we focus on the comparison of incompatible APIs gathering between the collected tools and our proposed artifact. In addition, we also adopt the same approach focusing on the Android framework to harvest incompatible APIs (as CiD^n in Section 5.6.1) since the API summary is no longer provided in the AOSP source set.

- IctApiFinder, even though the module of compatibility issue detection has been updated, the module of incompatible API extraction is still the same as the one published in the corresponding paper. To retrieve incompatible APIs, the developers extracted APIs from released SDKs API level 4 to API level 27. The SDK contains all of the APIs necessary for App development provided by Google. Our tool extracts only from the Android framework. Thus, the APIs not in the framework are missed by *AndroMevol* (cf., 5.6.2.1).

RQ5 Findings

Compared with the state-of-the-art detection tools in collecting incompatible APIs, *AndroMevol* outperforms all of them by harvesting at least eight times more incompatible APIs, including both device-specific and evolution-induced ones and 195,883 previously unreported ones.

[14] https://cs.android.com/android/platform/superproject/+/master:packages/modules/Wifi/framework/java/android/net/wifi/p2p/WifiP2pDevice.java

5.6 Discussion

We now discuss the key implications of this research, including prioritized research directions that should be conducted for mitigating the fragmentation impact on the Android community. Our literature review and experimental findings raise a number of issues and opportunities for research and practice communities.

5.6.1 Implication

Comprehensive Compatibility Issues Detection: Our systematic literature review revealed that app compatibility issues are induced by five types of incompatible APIs. Existing detection approaches all focus on compatibility issues induced by some specific types of incompatible APIs. To pinpoint issues induced by all types of incompatible APIs, customers need to resort to different detection tools, which makes the detection laborious and arduous. Therefore, we argue for a comprehensive issue detection approach, which could detect issues induced by all types of incompatible APIs.

Continuous Improvement to Adapt to the Fast Evolution of the Mobile Ecosystem. With the rapid evolution of the open-source Android Operating System, detection tool maintainers need to take new system releases into account. Many device vendors release lots of different models as their own publishing step. To detect the newly introduced compatibility issues, these tools need to be refined once a new system version is released and a new device induced. However, these tools are not self-adaptive. They all need to be carefully adjusted.

As an example towards demonstrating the necessity to continuously update the tools to adapt to the fast evolution of the mobile ecosystem, we spend additional efforts to update the open-source CiD project by extending its supported API ranges from 1-25 to 1-31 (Android12 with API level 31 is the latest Android release). The updated version is referred to as CiD^n. We then apply CiD^n to analyze the apps in Dataset2. Figure 5.19 summarizes the experimental results, along with that achieved by the original CiD. Clearly, CiD's performance has indeed been improved after adapting to the latest release of Android frameworks. This evidence strongly suggests the necessity to keep adapting compatibility issues detection tools to support the latest changes of the mobile ecosystem. We therefore argue that different automation approaches are needed

to facilitate the extraction of Android APIs in order to automate issue detection when new Android versions and devices are released.

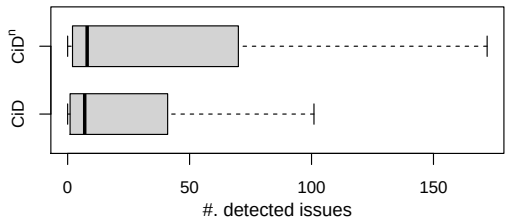


FIGURE 5.19: Comparison between original CiD and API life-cycle extended CiD.

Integrating Dynamic Testing to Verify Compatibility Issues: Currently, most research approaches proposed to tackle compatibility issues in Android apps rely on static analysis. However, efficient, static analysis is also known to yield many false-positive results. We argue that dynamic testing approaches should also be included to supplement the analysis of static analysis approaches (e.g., to practically verify the results yielded by static analysis approaches). It is nevertheless non-trivial to build a comprehensive dynamic testing environment for checking compatibility issues, as it needs to include all publicly available Android devices, for which the number is also continuously changing. To cope with this, we argue that crowdsourced mobile app testing could be leveraged, especially in lightweight mode directly supported by the Android system, to pinpoint and subsequently mitigate compatibility issues.

Characterizing Semantics-changing Incompatible APIs: In addition to the five types of incompatible APIs discussed in this work, which are all related to the existence of the APIs, there is another type of API-induced compatibility issue that goes beyond APIs' existence to concern their semantic changes. Given an API with semantic changes, even if its signature persisted in the framework, the client apps accessed into it could also be impacted. Such semantics changes will be propagated to the client app, which may not have yet adapted to such changes. As recently revealed by Liu et al. [155], there are indeed a number of Android APIs involving semantic changes during the evolution of the framework. However, such semantic changes are hard to be automatically identified, so as to the corresponding compatibility issues. Therefore, we argue that our community should also pay special attention to semantics-changed incompatible APIs and invent advanced approaches to mitigate them, either by carefully (1) documenting them if it is unavoidable to change the semantics of existing APIs or

(2) testing the client apps to identify and fix such issues before publishing the apps to end-users.

Supporting Automated Compatibility Issue Repair: Finally, after API compatibility issues are identified, we argue that automated approaches are also needed to help developers fix them [156]. This is especially true for such apps that have already been released to the public, as users may not even be able to install the apps or face runtime crashes even if the apps can be successfully installed. Automated repairing approaches could keep users from encountering such unfavorable situations, meanwhile helping app developers fix the issues for better future releases.

Supporting Issue Detection Besides Android Ecosystem: Compatibility issues exist in all sorts of different systems besides Android, such as the highly evolved Linux and their third-party customizations as well as the web browser systems. Different APIs are provided along with their system release. To enhance the robustness of such systems, developers should detect such issues as much as possible before their system's release. Our proposed approach could be extended to gather incompatible APIs, including the evolution-induced APIs and device-specific ones in order to do issue detection as the general issue detection approach always starts with incompatible APIs harvest. Our approach could facilitate researchers to focus on detailed algorithms to detect compatibility issues with the help of the collected incompatible APIs.

5.6.2 Threats to Validity

5.6.2.1 External Validity

There are several threats to validity associated with the results we presented in our replication study. One threat is the configuration of all our selected detection tools. All selected tools are implemented on top of the Soot static analysis framework, which also requires Android frameworks as input parameters. However, the version of the Soot and Android frameworks accessed in the selected artifacts may be still different because we cannot know the exact versions leveraged in every detection tool. Different Soot versions were released with different bug fixes and enhancement modules merged. They could have different performances and give different analysis results,

breaking the analysis validity. To mitigate this threat, we meticulously align the configurations among them as much as we can to provide approximately the same environment. Another threat depends on the approach to harvest incompatible APIs, especially between IctApiFinder and CiD. IctApiFinder extracts APIs from Android framework API levels 4 to 27 based on published artifacts, while CiD acquires from the source code of Android framework API levels 1 to 25 on their approach. Different ranges of API levels and the trade-offs made while pinpointing incompatible APIs would unavoidably bring in discrepancies, which may result in different performances even on the same dataset. In addition, our approach focuses on the signature of the APIs while CIDER could identify APIs with semantic changes, which are missed by *AndroMevol*.

The major threat to the validity of the work to extract a complete set of incompatible APIs is related to the selection of vendors with customized Android frameworks, which may not be representative of the whole ecosystem as there are many more vendors available in the ecosystem. Nevertheless, we have attempted to mitigate this impact by focusing on the most popular Android brands. Furthermore, because of various challenges put on by both Android and the customized vendors, we cannot locate the correct ROMs on the internet, although we have spent a significant amount of time doing that. Even for identified ROMs, we may not be able to extract the framework code for some vendors' framework platform versions, making the results incomplete. The corresponding results do not reflect the whole set of incompatible APIs in the wild. In addition, we focus the API extraction on the package framework.jar, which does not contain all the available APIs (such as some incompatible ones only identified by CIDER, IctApiFinder, and CiD, etc.). That's because we cannot locate the other package names and locations providing other APIs among different vendor OS releases. However, we have made extra manual efforts to ensure that the results that can be computed are indeed correct.

5.6.2.2 Internal Validity

Since we want to include a complete evolutionary history of the official Android framework, which is essential to observe evolution-induced incompatible APIs, we resort to two different approaches to extract APIs (e.g., source code parsing and direct bytecode extraction). These two approaches may introduce inconsistencies as (1) we cannot locate the exact framework version (at the source code level) that is customized by the

third-party vendors, and (2) not all the source code files (e.g., systemui-related code is not included in the framework bytecode but compiled into an independent APK) are packaged to the final framework version embedded in real devices. To mitigate the potential impact, we collect APIs from the source code of the official Android releases and harvest APIs from the same API level decompiled framework.jar extracted from the official Android virtual machines provided in Android Studio. We collect the package names that exist in both source code and framework.jar and set the list of package names as an API filter (cf., 5.5.1.1). With the API filter, we exclude APIs that do not reside in our collected packages (i.e., the package names of the APIs do not exist in our collected package names). This process, unfortunately, could also exclude actual incompatible APIs. However, the number of excluded APIs is generally small, making this impact neglectable.

We only take the API signature (the method signature refers to the combinations of the method return type, the method name, and the method parameters type list, thus method's checked exceptions are ignored.) into consideration when we determine incompatible APIs (cf., 5.5.1) in our proposed *AndroMevol*, which would induce false negatives. Besides, the updates of the class hierarchy including member methods and fields movement among parent and child classes are also ignored in our implementation. However, the movements of the APIs (methods/fields) among parents and their child classes can be and should be handled by the detection tools, such as the implementation of the detection tool, CiD, so as to avoid an explosion of the number of incompatible APIs. If we add all APIs belonging to parents' class to children's class, the number of incompatible APIs would increase sharply. To have a fair number of incompatible APIs and take full advantage of detection tools, we hand over the capability to manage class hierarchy to detection tools. In the future, we would enhance or develop our artifacts to consider these limitations to systematically detect as many types of compatibility issues as possible.

5.7 Related Work

In recent years, compatibility issues have been a hot topic in the Android community [20, 131, 156–159]. Since the apps are inseparable from the official Android APIs,

it is essential to probe compatibility issues caused by the evolution of the Android operating systems.

Besides the tools we investigated in the paper, there are many other works handling various API issues. For example, Li et al. [15, 83] build a prototype tool, CDA, to characterize deprecated Android APIs by mining the evolution of the Android framework. Similar method has also been applied to characterize inaccessible APIs [16] and inconsistent release time of Android apps [82]. Scalabrino et al. [20] introduce ACRYL, learning from the change histories of other apps in response to API evolution. It can identify compatibility issues, yet in addition suggest repairs. The authors empirically compare ACRYL and CiD and track down no obvious winner, but the results indicate the possibility of combining the two methods in the future. Later on, they extend their work [72] by enlarging the datasets and adding some interviews and details, but there is no obvious improvement in terms of the detection approach. Xia et al. [21] conduct a large-scale study on the practice of handling OS-induced API compatibility issues and their solutions, and they propose a tool named RAPID to ascertain whether a compatibility issue has been resolved. Mobilio et al. [70] acquaint a tool named FILO which can assist Android designers in tackling backward compatibility issues caused by API upgrades. FILO is designed to recognize app methods that need to be altered to adapt to the API changes and report symptoms observed in failed executions to facilitate repair. Mahmud et al. [24] propose ACID, an approach to detecting compatibility issues caused by API evolution. Experimental results demonstrate that ACID is more accurate and faster in detecting compatibility issues than previous techniques. The fly in the ointment is that ACID only considers the changes in Android method invocations and callbacks brought about by evolution rather than considering device-specific compatibility issues.

To detect such compatibility issues, different information flows are needed to identify by constructing inter- and intra-procedural control flow graph [160]. Qiu et al. [161] did an extensive comparison among three most prominent static analysis tools including FlowDroid [162] combined with IccTA [4], DroidRA [163, 164], AmanDroid [165, 166], and DroidSafe [167]. They spotted out the advantages and shortcomings of each tools and revealed that it is important to provide detailed configuration and setup environment specification to guarantee the replicability of experiments.

In non-Android communities, research on compatibility issues is also pervasive [168–172]. Sawant et al. [172] analyze clients of popular third-party Java APIs and the JDK API and publicise a large dataset; also, they look into the connection between the client's response patterns and the deprecation policy the related API adopted. Chen et al. [173] present an approach named DeBBI, which leverages the test suites of various client projects to detect library behavioral backward incompatibilities.

To compare different tools developed for the same issue, Su et. al. [174] did an extensive comparison and proposed a new benchmark called Themis facilitating our research community for automated GUI testing. They collected critical bugs reported on Github with respect to their bug label revealing the severity and did experiments with five state-of-the-art testing tools integrated with Monkey [175], and then gave out qualitative and quantitative analysis result. They successfully identified 5 different challenges that these tools still face, such as the reachability of deep use scenarios, test input generation etc., and shed lights on future research based on their systematic analysis results, such as integrating heuristics to improve the capability to spot GUI bugs.

5.8 Conclusion

In this paper, we have conducted a literature review on research targeting Android app compatibility issues. Based on this review, we are able to identify nine state-of-the-art works proposed to detect compatibility issues in Android apps and among which we have summarized five types of incompatible issues reported by our fellow researchers. We then confirm the reproducibility of the selected tools based on a replication study by running the tools against their original datasets. We further go one step deeper to conduct an empirical comparison study among the selected tools. Our findings indicate that compatibility issues detection is still at an early stage, which requires attention from the community to keep improving so as to achieve sound compatibility issues detection. As categorised and reported by other researchers, there are five types of incompatible APIs available in the mobile ecosystem but none of the existing harvest approaches is capable of collecting all of them. To fill this gap, in this work, we propose to introduce to the community a novel prototype called *AndroMevol*, which endeavors to construct a list including as many incompatible APIs as possible. Experimental results demonstrate

that *AndroMevol* is effective in achieving its purpose to generate a list of incompatible APIs, which are useful for supporting the detection of compatibility issues in real-world Android apps.

Chapter 6

Conclusion and Future Work

6.1 Key Contributions

This Ph.D. work focuses on the characterization and detection of Android app compatibility issues. In detail, we investigated two types of inconsistent APIs (i.e., special supply in particular OS versions or OS customizations) from the official Android releases and third-party OS customizations, which could eventually induce compatibility issues in Android apps. Consequently, we explored different approaches for compatibility issue identification.

First, **we proposed an inconsistent API detection tool, AndroSea**, that identifies functions with the same signatures but different implementations between two consecutive Android code bases. Such functions are referred to as Silently Evolved Methods (SEMs). The development of Android applications relies heavily on the APIs extracted from the Android code base and provided via development documentation by Google. Aside from the deletion and addition of APIs in the newer release of Android OS, the SEMs, even though offered to developers in continuous Android releases, could still induce compatibility issues. However, it is still unknown the distribution of SEMs among Android releases. To fill this research gap, we proposed AndroSea to pinpoint SEMs in two consecutive Android releases. It first pre-processes the Android code base with a manually curated whitelist to filter out API irrelevant modules because those modules are implemented for unit tests, command-line tools, Android Asset Packaging tools, etc. It then extracts APIs and their implementations from the remaining modules and

compares APIs and their implementation for two continuous Android releases. With the help of AndroSea, we could successfully identify 4,769 SEMs across ten consecutive official Android releases, and 2,271 of them are publicly provided to app developers to implement their function intention while the other SEMs could still be invoked with other circumvent approaches, such as Java reflection, by app developers. To determine if semantic changes are involved in our identified SEMs, we manually analyzed a statistically significant sample containing 562 SEMs and found that 363 of them contain semantic changes. We also investigated the utilization of these curated SEMs in a sample of 1,000 Android apps and observed that 957 of these apps call at least one SEM, which indicates that SEMs are widespread invoked in apps. Moreover, the invocations of SEMs with appropriate guard checks were also spotted in our selected sample apps, which demonstrates that SEMs could induce compatibility issues. That is because developers introduce guard checks before the invocation of SEMs to fix the potential incompatibility problem. In the study, we (i) identified SEMs across different versions of the Android API, (ii) reported the characteristics of these methods, and (iii) analyzed the impact that SEMs might have on real-world apps.

Secondly, **an extensive empirical study was conducted** to investigate another type of inconsistent API, which is introduced in third-party OS customizations. The open-source nature of Android OS enables third-party OS providers and device vendors to release their special OS customizations to satisfy customers' proprietary needs. Instead of constructing their customizations from scratch, device vendors and OS providers always modify their copy of the official Android OS code base. They not only add new features to the code base but also modify or even delete existing modules in the code base, which could induce inconsistencies. In addition, they also need to periodically perform merge operations based on the official Android OS to incorporate new bug fixes, security patches, and new functionalities. These merge operations could always lead to merge conflicts. However, it is still unclear to us how these OS customizations' maintainers resolve these conflicts and the impact of these conflicts on the installable APKs. To bridge this gap, we did an extensive empirical study on eight open-source Android OS customizations and investigated how their developers incorporate new changes from the Android OS releases in the customizations, and even the impact of the merge conflicts on the final Android apps (randomly sampled 1,000

Android apps). By reproducing the merge operations of the OS customizations, we revealed that (1) developers performed this type of merge operation for 9.7% of versions released by the Android OS, (2) developers have encountered at least one conflict in 41.3% of the merge operations performed, (3) source code methods are the code entities most affected by the conflicts, (4) 58.1% of the conflicts required developers to change the customized OSs, and (5) 64.4% of the considered apps use at least one method affected by a conflict.

Third, **a replicability study was carried out** to unveil the advantages and disadvantages of the existing compatibility issue detection tools and to guide future research. Different types of inconsistent APIs were detected, including those discussed previously, as they are widespread in Android app development. Without proper treatment for these APIs when implementing function intentions in app development, compatibility issues reflected by the runtime crashes, the error message “NoSuchMethodError”, etc., would be encountered by app users, hindering them from finishing their tasks, lowering their expectations for the apps, or even causing them to remove and download alternatives. To relieve the burden on app developers, a great deal of research has been conducted to analyze the compatibility issues in Android apps. However, the landscape of compatibility analysis in the Android ecosystem is still unclear. Particularly, it is transparent for researchers and app developers how many approaches in issue identification are available and what their strengths and weaknesses are when applied to issue detection in real-world Android apps. Furthermore, it is also unclear to what extent can we reproduce their experimental results and how well do the tools compare with each other in terms of detecting compatibility issues. To answer these research concerns, a systematic literature review was first conducted among 19, and 44 issue-detection-related publications were successfully identified, of which 9 proposed automatic approaches to spot compatibility issues. We summarized the conventional working process of compatibility issue detection as two separate steps, harvesting incompatible APIs first and determining compatibility issues by checking if the invocation of the incompatible API has proper guard checks or not. Moreover, we classified the API-induced compatibility issues into five types, including Evolution-induced (Method), Evolution-induced (Field), Device-specific (Method), Device-specific (Field), and Override/Callback according to the origin of the underlying inconsistent APIs. Unfortunately, none of the existing approaches can tackle all five types of API-induced compatibility

issues. We replicated all the identified approaches following the instructions presented in the papers and confirmed that most of the experimental results could be reproduced. Comparison between the selected tools concerning two common datasets revealed that (1) compatibility issues detection approaches that achieve their purpose via systematically harvested incompatible API rules (such as CiD and IctApiFinder) can identify significantly more issues than those having their rules summarized manually, and (2) the intersection among the results reported by the selected tools is relatively small. To provide a systematic incompatible API collection approach, we proposed a tool, *AndroMevol*, to systematically harvest incompatible APIs from five popular OS vendors. Eventually, we collected a total of 397,678 incompatible APIs including 195,883 previously untouched ones, which could be accessed by Android apps inducing compatibility issues.

6.2 Limitations

This research works on compatibility issues in the Android ecosystem. We proposed tools and conducted studies to investigate and characterize the compatibility issues concerning our curated datasets. Although our research unveils several interesting findings and provides promising guidelines for researchers and practitioners, it has some limitations that we need to discuss.

First of all, we acknowledge that the term *compatibility* has a huge context, and various potential variations have been discussed in the existing literature. To have a clear research background and distinct context, we only focus on app compatibility as defined in the Google Android Developers reference site [5].

To identify silently evolved methods, we proposed a detection tool, AndroSea, and performed it on ten official consecutive Android OS releases, spanning from Android API level 19 to API level 29 (API level 20 was removed as it was reserved for wearables). However, the latest API release is API level 33, and the OS releases fed to AndroSea are a bit old. The market share of the earlier OS releases is becoming less and less and the research on newer ones is also more meaningful. The rapid evolution of Android requires us to redo experiments by incorporating new OS releases and update the conclusions timely. Fortunately, AndroSea is open for new OS releases. Feeding new

releases of Android to ANDROSEA redoing experiments in the future would advance our research on identifying Silently Evolved Methods.

To characterize the evolution of the customized Android OS, we curated 8 open-source OS customizations. However, some OS customization projects were canceled and are not available for practitioners and researchers anymore. Additionally, for continuously maintained ones, certain git operations (e.g., git rebase) would make the merge operation involving updates from official OS releases unavailable for researchers. To have a better understanding of the co-evolution between OS customizations and official releases, it is necessary to collect newly available open-source customizations and conduct a detailed analysis of merge operations. The pipeline adopted in our research is transparent to the customizations. Collecting currently available open-source customizations and performing the same analysis pipeline on them would enhance our research in characterizing OS customizations.

To summarize the strengths and weaknesses of the currently available issue detection approaches, we conducted a systematic literature review to gather issue detection tools and presented a replicability study concerning the original and an additional open-source dataset. To improve the ability to identify compatibility issues, we proposed an incompatible API harvesting tool, *AndroMevol*, to collect incompatible APIs from five popular Android brands. However, new advanced approaches are published to overcome the limitations of previously released tools and *AndroMevol* only takes five additional popular brands into account, which weakens our research. To improve the validity of our research, we plan to include the newly released detection tools to summarize the strengths and weaknesses of all available tools and many more famous Android brands to harvest incompatible APIs, and even propose our approach to identify issues based on the curated complete incompatible APIs.

To evaluate the effectiveness of the proposed approaches, we selected different app datasets from different data sources, such as AndroZoo [176], AndroidCompass [123]. The detailed Android apps used in the experiments are listed as follows:

- 1000 randomly collected closed-source Android apps from AndroZoo in Section 3.4.4.

- 1000 randomly collected closed-source Android apps from AndroZoo in Section 4.4.
- 65 open-source Android apps collected from published works for replicability study in Section 5.4.2.
- 645 open-source Android apps collected from AndroidCompass in Section 5.4.2 [1].

All of the above datasets are curated independently. Therefore, no potential overlap exists between them. The discrepancies between the experimental datasets potentially undermine the validity and limit the extensibility of the proposed approaches. However, we curated the datasets from AndroZoo randomly, and the open-source datasets from existing literature, which would mitigate the potential threats saddled by the different experimental datasets.

6.3 Future Work

We have identified several key research directions based on what we have achieved.

More advanced issue-detection tools. Fragmentation has been a severe problem for the Android ecosystem for years. The widespread inconsistent APIs in Android OSs and the compatibility issues induced in Android apps have attracted the attention of researchers. However, the existing approaches are developed only for the evolution API induced compatibility issues. They are not only no longer maintained but also incapable of detecting other types of issues. Besides, these tools do not possess the adaptive capacity to consider new versions of the official OS and their corresponding customizations, released by OS providers and device vendors. We, therefore, argue for new self-adaptive approaches that can detect all kinds of compatibility issues and consider new Android OS customizations.

Ground truth and evaluation for compatibility issue detection tools. By convention, the effectiveness of issue-detection tools is measured on a small dataset of Android apps. Tool developers always perform issue detection approaches on Android

[1] Actually, there are 1,375 Android apps in AndroidCompass, but we could only successfully build 645 of them. Thus, we finally curated 645 open-source Android apps.

apps and report the identified issues to app developers to confirm the true positive ones. This is time-consuming and hard to reproduce once the app developers fix the reported issues. It is also challenging to determine valid accuracy and precision for the detection tools, as the ground truth of Android apps containing a specific number of compatibility issues is still missing. To have a fair comparison for issue detection tools, we advocate an open-source ground truth of Android apps for compatibility issue research. Additionally, the detection tools are only evaluated by researchers, and it is unknown whether they are used by app developers to identify issues and fix them before app release. Thus, we suggest involving app developers with various levels of experience to measure the usability of the existing detection tools and collect key requirements from them to guide the development of issue detection tools in the future.

The characterization of issues among Android apps. Although many approaches have been proposed to identify compatibility issues in Android apps, little attention has been paid to the evolution and distribution of different compatibility issues in Android apps: which types of issues are the most common ones? Do different types of Android apps suffer different impacts in terms of different types of issues? Do developers handle detected issues as they release newer app versions? Answering these questions could help the community better understand the status quo of different types of compatibility issues among the released Android apps, focus on the development of issue detection tools, and propose new, more effective automatic approaches to fix these issues before release. Thus, we suggest a panoramic analysis to characterize compatibility issues in different types of Android apps, such as apps in different categories, different numbers of installations, different app markets, etc.

Automatic repair compatibility issues in Android apps. Identified issues require fixes to avoid runtime crashes, error message prompts, unsuccessful installations, etc. However, fixing the issues one by one is time-consuming for developers. To save time for developers and enable them to focus on critical function implementation, we argue that automated approaches are necessary for developers to fix compatibility issues. Automated repairing approaches could prevent users from encountering such unfavorable situations while speeding up more robust app releases.

Feedback collection from Android practitioners and users. Incompatible APIs exist between different Android versions and customizations. Are these incompatible APIs

inevitable when releasing new versions of official and customized Android? And also when implementing Android apps, do app developers always take the incompatible APIs into consideration? What do the app developers and OS maintainers think about incompatible APIs and the consequently induced compatibility issues? Besides, the most important app users' opinions with regard to compatibility issues have not been collected yet. By interviewing Android practitioners and users, researchers could gain a deeper understanding of the topic compatibility, which could provide for some new research directions and even foster moe radical solutions.

Automated potential issues fix using Large Language Models. The emergence of large language models provides a potential new solution to this problem. The nature of Android being open-source further makes the utilization of LLM (Large Language Model) more practical. Different versions of Android releases, Android customizations, open-source Android app implementations, along with open discussions on question-and-answer websites could all be used to train a LLM. With the neural knowledge provided by the LLM, developers could be promptly reminded when they try to invoke some potentially incompatible APIs, which could practically and significantly reduce the occurrences of compatibility issues in Android apps.

6.4 Summary

In summary, this Ph.D. project focuses on detecting and characterizing Android app compatibility issues. We proposed an automatic approach ANDROSEA to identify silently evolved methods across ten continuous official Android releases and conducted an extensive empirical study to investigate inconsistencies between official Android releases and their third-party customizations. We also replicated existing approaches for identifying inconsistencies-induced issues, summarized their strengths and weaknesses, and proposed *AndroMevol* to systematically harvest incompatible APIs to enhance issue detection approaches. Furthermore, we proposed several research directions, including the need for more advanced issue-detection tools that can adapt to new Android OS customizations and detect all kinds of compatibility issues, open-source ground truth and evaluation for compatibility issue detection tools, characterization of compatibility issues among Android apps, and automatic repair of compatibility issues

in Android apps. By addressing these research directions, we aim to provide device users with more robust Android apps.

Bibliography

[1] Android market share. https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/, 2023. Accessed: 28-02-2023.

[2] Android statistics. https://www.businessofapps.com/data/android-statistics/, 2023. Accessed: 28-02-2023.

[3] Mobile app download statistics & usage statistics (2023). https://buildfire.com/app-statistics/, 2023. Accessed: 28-02-2023.

[4] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[5] *Device compatibility overview*, 2023. URL https://developer.android.com/guide/practices/compatibility. Accessed: 22-08-2023.

[6] *Android Compatibility Program Overview*, 2023. URL https://source.android.com/docs/compatibility/13/android-13-cdd#31_managed_api_compatibility. Accessed: 22-08-2023.

[7] *Android Compatibility Program Overview*, 2023. URL https://source.android.com/docs/compatibility/overview. Accessed: 22-08-2023.

[8] Code names, tags, and build numbers. https://source.android.com/docs/setup/about/build-numbers, 2023. Accessed: 03-03-2023.

[9] Android version codes. https://developer.android.com/reference/android/os/Build.VERSION_CODES, 2023. Accessed: 03-03-2023.

[10] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in android apps. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–177. IEEE, 2018.

[11] Google. API reference. https://developer.android.com/reference, 2023. Accessed: 04-03-2023.

[12] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *ICSM*, pages 70–79, 2013.

[13] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change-and fault-proneness on the user ratings of android apps. *TSE*, 41(4): 384–407, 2014.

[14] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *ISSTA*, 2018.

[15] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Cda: Characterising deprecated android apis. *Empirical Software Engineering (EMSE)*, 2020.

[16] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *ICSME*, pages 411–422. IEEE, 2016.

[17] Google. About the Android Open Source Project. https://source.android.com, 2023. Accessed: 04-03-2023.

[18] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability (TRel)*, 2019.

[19] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237, 2016.

[20] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE, 2019.

[21] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. How android developers handle evolution-induced api compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 886–898. IEEE, 2020.

[22] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. A large-scale study of application incompatibilities in android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 216–227, 2019.

[23] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 878–888. IEEE, 2019.

[24] Tarek Mahmud, Meiru Che, and Guowei Yang. Android compatibility issue detection using api differences. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 480–490. IEEE, 2021.

[25] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *FSE*, pages 477–487, 2013.

[26] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94, 2014.

[27] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th working conference on mining software repositories*, pages 2–11, 2014.

[28] Patrick PK Chan and Wen-Kai Song. Static detection of android malware by using permissions and api calls. In *2014 International Conference on Machine Learning and Cybernetics*, volume 1, pages 82–87. IEEE, 2014.

[29] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow apis and machine learning. *Information and software technology*, 75:17–25, 2016.

[30] Guanhong Tao, Zibin Zheng, Ziying Guo, and Michael R Lyu. Malpat: Mining patterns of malicious and benign android apps via permission-related apis. *IEEE Transactions on Reliability*, 67(1):355–369, 2017.

[31] Shishuai Yang, Rui Li, Jiongyi Chen, Wenrui Diao, and Shanqing Guo. Demystifying android non-sdk apis: measurement and understanding. In *Proceedings of the 44th International Conference on Software Engineering*, pages 647–658, 2022.

[32] Yi He, Yacong Gu, Purui Su, Kun Sun, Yajin Zhou, Zhi Wang, and Qi Li. A systematic study of android non-sdk (hidden) service api security. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[33] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 204–215, 2019.

[34] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automated deprecated-api usage update for android apps: How far are we? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 602–611. IEEE, 2020.

[35] Stefanus A Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automatic android deprecated-api usage update by learning from single updated example. In *Proceedings of the 28th international conference on program comprehension*, pages 401–405, 2020.

[36] Stefanus A Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. Androevolve: Automated

update for android deprecated-api usages. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 1–4. IEEE, 2021.

[37] Stefanus A Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. Androevolve: Automated android api update with data flow analysis and variable denormalization. *Empirical Software Engineering*, 27(3):73, 2022.

[38] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634, 2013.

[39] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.

[40] Roberto Gallo, Patricia Hongo, Ricardo Dahab, Luiz C Navarro, Henrique Kawakami, Kaio Galvão, Glauber Junqueira, and Luander Ribeiro. Security and system architecture: Comparison of android customizations. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–6, 2015.

[41] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting inconsistent security configurations in custom android roms via differential analysis. In *USENIX Security Symposium*, pages 1153–1168, 2016.

[42] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[43] Roee Hay. fastboot oem vuln: Android bootloader vulnerabilities in vendor customizations. In *WOOT*, 2017.

[44] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A gray-box android fuzzer for vendor service customizations.

In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11. IEEE, 2017.

[45] Domenico Cotroneo, Antonio Ken Iannillo, and Roberto Natella. Evolutionary fuzzing of android os vendor system services. *Empirical Software Engineering*, 24:3630–3658, 2019.

[46] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *Proceedings of The Web Conference 2020*, pages 3063–3069, 2020.

[47] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 87–102. IEEE, 2021.

[48] Dongsong Yu, Guangliang Yang, Guozhu Meng, Xiaorui Gong, Xiu Zhang, Xiaobo Xiang, Xiaoyu Wang, Yue Jiang, Kai Chen, Wei Zou, et al. Sepal: Towards a large-scale analysis of seandroid policy customization. In *Proceedings of the Web Conference 2021*, pages 2733–2744, 2021.

[49] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. Large-scale security measurements on the android firmware ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1257–1268, 2022.

[50] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.

[51] Mehran Mahmoudi and Sarah Nadi. The android update problem: An empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 220–230, 2018.

[52] Chungha Sung, Shuvendu K Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 172–181, 2020.

[53] Max Ellis, Sarah Nadi, and Danny Dig. Operation-based refactoring-aware merging: An empirical evaluation. *IEEE Transactions on Software Engineering*, 2022.

[54] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering*, pages 83–92. IEEE, 2012.

[55] Hammad Khalid, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. Prioritizing the devices to test your app on: A case study of android game apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 610–620, 2014.

[56] Xuan Lu, Xuanzhe Liu, Huoran Li, Tao Xie, Qiaozhu Mei, Dan Hao, Gang Huang, and Feng Feng. Prada: Prioritizing android devices for apps by mining large-scale usage data. In *Proceedings of the 38th International Conference on Software Engineering*, pages 3–13, 2016.

[57] Mattia Fazzini and Alessandro Orso. Automated cross-platform inconsistency detection for mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 308–318. IEEE, 2017.

[58] Malinda Dilhara, Haipeng Cai, and John Jenkins. Automated detection and repair of incompatible uses of runtime permissions in android apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 67–71, 2018.

[59] Daoyuan Wu, Ximing Liu, Jiayun Xu, David Lo, and Debin Gao. Measuring the declared sdk versions and their consistency with api calls in android apps. In *Wireless Algorithms, Systems, and Applications: 12th International Conference, WASA 2017, Guilin, China, June 19-21, 2017, Proceedings 12*, pages 678–690. Springer, 2017.

[60] Daoyuan Wu, Debin Gao, and David Lo. Scalable online vetting of android apps for measuring declared sdk versions and their consistency with api calls. *Empirical Software Engineering*, 26:1–32, 2021.

[61] Cong Li, Chang Xu, Lili Wei, Jue Wang, Jun Ma, and Jian Lu. Elegant: Towards effective location of fragmentation-induced compatibility issues for android

apps. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 278–287. IEEE, 2018.

[62] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8, 2012.

[63] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, pages 245–251. Springer, 2011.

[64] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[65] Todd J Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry: Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, pages 1–8. Springer, 2012.

[66] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 532–542, 2018.

[67] Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. Analyses for specific defects in android applications: a survey. *Frontiers of Computer Science*, 13:1210–1227, 2019.

[68] Ziyi Zhang and Haipeng Cai. A look into developer intentions for app compatibility in android. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 40–44. IEEE, 2019.

[69] Mariano Ceccato, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orrù, and Paolo Tonella. Toward in-vivo testing of mobile applications. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 137–143. IEEE, 2019.

[70] Marco Mobilio, Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. Filo: Fix-locus localization for backward incompatibilities caused by android framework upgrades. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1292–1296, 2020.

[71] Isabel K Villanes, Andre Takeshi Endo, and Arilo C Dias-Neto. Using app attributes to improve mobile device selection for compatibility testing. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, pages 31–39, 2020.

[72] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Valentina Piantadosi, Michele Lanza, and Rocco Oliveto. Api compatibility issues in android: Causes and effectiveness of data-driven detection techniques. *Empirical Software Engineering*, 25(6):5006–5046, 2020.

[73] Chen Xu, Yan Xiong, Wenchao Huang, Zhaoyi Meng, Fuyou Miao, Cheng Su, and Guangshuai Mo. Identifying compatibility-related apis by exploring biased distribution in android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 280–281, 2020.

[74] Demetrio Guilardi, Jalves Nicácio, Bianca M Napoleão, and Fabio Petrillo. Are apps ready for new android releases? In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 66–76, 2020.

[75] Tarek Mahmud, Mujahid Khan, Jihan Rouijel, Meiru Che, and Guowei Yang. Api change impact analysis for android apps. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 894–903. IEEE, 2021.

[76] Tarek Mahmud, Meiru Che, and Guowei Yang. Analyzing the impact of api changes on android apps. *Journal of Systems and Software*, page 111664, 2023.

[77] eMarketer. eMarketer Unveils New Estimates for Mobile App Usage. https://www.emarketer.com/Article/eMarketer-Unveils-New-Estimates-Mobile-App-Usage/1015611, 2020. Last updated: 11-Apr-2017.

[78] statista. Mobile operating systems' market share worldwide from January 2012 to July 2020. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009, 2020. Last updated: 17-Aug-2020.

[79] statista. Number of available applications in the Google Play Store from December 2009 to June 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store, 2020. Last updated: 17-Aug-2020.

[80] Mark D. Syer, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal*, page 485–508, 2015.

[81] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. How do android operating system updates impact apps? In *MobileSoft*, pages 156–160, New York, NY, USA, 2018. ACM.

[82] Li Li, Tegawendé Bissyandé, and Jacques Klein. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *ISSRE*, pages 212–223. IEEE, 2018.

[83] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *MSR*, 2018.

[84] Google. API reference. https://developer.android.com/reference, 2020. Last updated: 15-Sep-2020.

[85] Google. ConnectivityManager. https://developer.android.com/reference/android/net/ConnectivityManager#getAllNetworkInfo(), 2020. Last updated: 29-Sep-2020.

[86] MySQL. SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements. https://dev.mysql.com/doc/refman/8.0/en/savepoint.html, 2020. Last updated: 29-Sep-2020.

[87] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *CETUS*, 2011.

[88] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

[89] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ASE*, 2014.

[90] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Android malware detection using deep learning on api method sequences. *arXiv preprint arXiv:1712.08996*, 2017.

[91] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[92] Zhuo Ma, Haoran Ge, Yang Liu, Meng Zhao, and Jianfeng Ma. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE access*, 7:21235–21245, 2019.

[93] statista. Share of global smartphone shipments by operating system from 2014 to 2023, 2021. URL https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems.

[94] verge-android. There are now 2.5 billion active android devices, 2021. URL https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-numberstatistic-keynote.

[95] aosp. About the android open source project, 2021. URL https://source.android.com.

[96] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 204–215, New York, NY, USA, 2019. Association for Computing Machinery.

[97] Python-customized-scripts. Customized android: Dataset and exploratory scripts, 2021. URL https://zenodo.org/record/6272071.

[98] LineageOS. Lineageos, 2021. URL https://lineageos.org.

[99] Daniel R Thomas, Alastair R Beresford, and Andrew Rice. Security metrics for the android ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 87–98, 2015.

[100] Daniel R Thomas, Alastair R Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. The lifetime of android api vulnerabilities: case study on the javascript-to-java interface. In *Cambridge International Workshop on Security Protocols*, pages 126–138. Springer, 2015.

[101] Wikipedia. List of custom android distributions, 2021. URL https://en.wikipedia.org/wiki/List_of_custom_Android_distributions.

[102] eproject. /e/, 2021. URL https://e.foundation.

[103] AOKP user bases. Aokp user bases, 2013. URL https://www.androidpolice.com/2013/09/28/aokp-rom-passes-3-5-million-users-nexus-7-2013-flo-and-oppo-find-5-android-4-3-

[104] LineageOS-GitHUb. Lineageos github, 2021. URL https://github.com/LineageOS.

[105] *Codenames, Tags, and Build Numbers*, 2021. URL https://source.android.com/setup/start/build-numbers.

[106] *Java parser*, 2021. URL https://javaparser.org/.

[107] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017.

[108] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. Predicting merge conflicts in collaborative software development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[109] *Xiaomi products*, 2021. URL https://en.wikipedia.org/wiki/List_of_Xiaomi_products.

[110] *OnePlus products*, 2021. URL https://en.wikipedia.org/wiki/OnePlus.

[111] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N Nguyen. Detecting semantic merge conflicts with variability-aware execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 926–929, 2015.

[112] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 190–200, 2011.

[113] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 168–178, 2011.

[114] Danhua Shao, Sarfraz Khurshid, and Dewayne E Perry. Sca: a semantic conflict analyzer for parallel changes. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 291–292, 2009.

[115] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 1 (OOPSLA):1–27, 2017.

[116] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. Understanding semistructured merge conflict characteristics in open-source java projects (journal-first abstract). In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 955–955, 2018.

[117] Gleiph Ghiotto, Leonardo Murta, Marcio Barros, and Andre Van Der Hoek. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, 46(8):892–915, 2018.

[118] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. Intellimerge: a refactoring-aware software merging technique. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[119] Yuichi Nishimura and Katsuhisa Maruyama. Supporting merge conflict resolution by using fine-grained code change history. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 661–664. IEEE, 2016.

[120] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of api evolution literature. *ACM Computing Surveys (CSUR)*, 54(8):1–36, 2021.

[121] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[122] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and software technology*, 64:1–18, 2015.

[123] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. Androidcompass: A dataset of android compatibility checks in code repositories. *arXiv preprint arXiv:2103.09620*, 2021.

[124] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. Automatically detecting api-induced compatibility issues in android apps: A comparative analysis (replicability studies). In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*, 2022.

[125] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4): 571–583, 2007.

[126] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. Deep learning for android malware defenses: a systematic literature review. *arXiv preprint arXiv:2103.05292*, 2021.

[127] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. Research on third-party libraries in android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering*, 2021.

[128] Md. Shamsujjoha, John Grundy, Li Li, Hourieh Khalajzadeh, and Qinghua Lu. Developing mobile applications via model driven development: A systematic literature review. *Information and Software Technology (IST)*, 2021.

[129] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 2018.

[130] Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. Mimic: Ui compatibility testing system for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 246–256. IEEE, 2019.

[131] Haoyu Wang, Hongxuan Liu, Xusheng Xiao, Guozhu Meng, and Yao Guo. Characterizing android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 280–292. IEEE, 2019.

[132] Acid. https://github.com/TSUMahmud/acid, 2021.

[133] Acryl. https://github.com/intersimone999/acryl, 2021.

[134] Download pivot. https://ficissuepivot.github.io/Pivot/, 2021.

[135] Cid. https://github.com/lilicoding/CiD, 2021.

[136] Ictapifinder. https://github.com/DongjieHe/IctApiFinder, 2021.

[137] Cider. https://github.com/cideranalyzer/cideranalyzer.github.io, 2021.

[138] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering*, 46(11):1176–1199, 2018.

[139] Ficfinder project homepage. http://sccpu2.cse.ust.hk/ficfinder/, 2021.

[140] *OPPO's share of smartphone shipments worldwide*, 2021. URL https://www.statista.com/statistics/628545/global-market-share-held-by-oppo-smartphones/.

[141] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.

[142] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[143] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[144] Gadgetbridge. https://github.com/Freeyourgadget/Gadgetbridge, 2022.

[145] Ankidroid. https://github.com/ankidroid/Anki-Android, 2022.

[146] Libretorrent. https://github.com/proninyaroslav/libretorrent, 2022.

[147] Mozstumbler. https://github.com/mozilla/MozStumbler, 2022.

[148] *Official Android API reference*, 2022. URL https://developer.android.com/reference.

[149] *Soot Framework*, 2022. URL http://soot-oss.github.io/soot/.

[150] Various firmware hosting site, 2022. https://firmwarefile.com/.

[151] Oppo firmware update site, 2022. https://support.oppo.com/au/software-update/.

[152] Xiaomi firmware update site, 2022. https://c.mi.com/global/miuidownload/index.

[153] *Sample Size Calculator*, 2022. URL https://www.surveysystem.com/sscalc.htm.

[154] Pei Liu, Mattia Fazzini, John Grundy, and Li Li. Do customized android frameworks keep pace with android? In *The 19th International Conference on Mining Software Repositories (MSR 2022)*, 2022.

[155] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. Identifying and characterizing silently-evolved methods in the android api. In *The 43rd ACM/IEEE International Conference on Software Engineering, SEIP Track (ICSE-SEIP 2021)*, 2021.

[156] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. Towards automatically repairing compatibility issues in published android apps. In *The 44th International Conference on Software Engineering (ICSE 2022)*, 2022.

[157] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415. IEEE, 2017.

[158] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference*, pages 601–614, 2018.

[159] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. Semantic patches for java program transformation (experience report). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[160] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[161] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186, 2018.

[162] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[163] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329, 2016.

[164] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. Taming reflection: An essential step towards whole-program analysis of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2020.

[165] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1329–1341, 2014.

[166] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[167] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[168] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[169] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260. IEEE, 2015.

[170] Jing Zhou and Robert J Walker. Api deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 266–277, 2016.

[171] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 360–369. IEEE, 2016.

[172] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

[173] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 112–124, 2020.

[174] Ting Su, Jue Wang, and Zhendong Su. Benchmarking automated gui testing for android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 119–130, 2021.

[175] Monkey. http://developer.android.com/tools/help/monkey.html, 2022.

[176] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.