Do Customized Android Frameworks Keep Pace with Android?

Pei Liu
pei.liu@monash.edu
Monash University
Melbourne, Victoria, Australia

Mattia Fazzini
mfazzini@umn.edu
University of Minnesota
Minneapolis, Minnesota, United States

John Grundy
john.grundy@monash.edu
Monash University
Melbourne, Victoria, Australia

Li Li*
li.li@monash.edu
Monash University
Melbourne, Victoria, Australia

ABSTRACT

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. 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 provide more insight into this critical aspect of the Android ecosystem, we present an empirical study that investigates how eight open-source customizations of the Android OS merge the changes from the Android OS into their projects. The study analyzes how often the developers from the customized OSs merge changes from the Android OS, how often the developers experience textual merge conflicts, and the characteristics of these conflicts. Furthermore, to analyze the effect of the conflicts, the study also analyzes how the conflicts can affect a randomly selected sample of 1,000 apps. After analyzing 1,148 merge operations, we identified that developers perform these operations for 9.7% of the released versions of the Android OS, developers will encounter at least one conflict in 41.3% of the merge operations, 58.1% of the conflicts required developers to change the customized OSs, and 64.4% of the apps considered use at least one method affected by a conflict. In addition to detailing our results, the paper also discusses the implications of our findings and provides insights for researchers and practitioners working with Android and its customizations.

*Corresponding author.

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3527963

ACM Reference Format:
Pei Liu, Mattia Fazzini, John Grundy, and Li Li. 2022. Do Customized Android Frameworks Keep Pace with Android?. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524842.3527963

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 [40] and running on more than two billion devices [44]. 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) [7]. 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, 21, 24]. 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 [9, 20, 23, 26, 30, 48]; 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 [29], 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 [9, 14, 20, 23, 26, 30, 48]). 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 [11, 17, 22, 37, 45, 47]) 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 [36].

2 TERMINOLOGY & MOTIVATION

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

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.

2.2 Motivation

Fig. 1 provides an example[1] of a merge conflict appearing in the LineageOS [27], an open-source customization of the Android OS that offers custom OS management and security features [42, 43]. 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 Fig. 1c) has a different implementation with respect to the same method in the upstream repository (lines 22-33 in Fig. 1c).

Downstream developers needed to handle such a conflict with careful attention because the part of the conflict from the upstream repository contains an update. Fig. 1a and 1b 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 Fig. 1b). Downstream developers identified these changes and suitably updated their implementation of the `mapIconSets` method as reported in Fig. 1d (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 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.

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

```
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 ...
```

(a) 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 ...
```

(b) 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 ...
```

(c) 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 ...
```

(d) Conflict resolution in the LineageOS.

Figure 1: Example of a merge conflict and its resolution in the LineageOS.

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?

Table 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/J3zLO	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

3.1 Dataset Selection

To identify relevant downstream projects, we analyzed a readily available and curated list of customized OSs based on Android [46]. 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/ [13]) 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 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 [6].

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.

Table 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	-

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 1 provides an example of a relevant merge commit from the LineageOS.

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

Listing 1: Commit merging changes from the Android OS.

The listing reports the identifier for the merge commit (`80d8b6467ec4` 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 1 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 [28].
Results: Table 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 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 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 2 also show that **downstream projects merge updates only for a small percentage (9.7%) of the versions released by the upstream projects**. Fig. 2 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 LineageOS are the projects with the highest number of merge operations in the recent years.

To further detail how developers have merged updates over time, Fig. 3 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

[2]We identified the tags from the Android OS documentation [1].

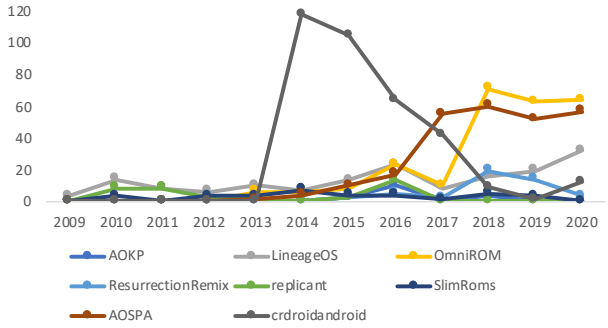

Figure 2: Distribution of merge commits over time.

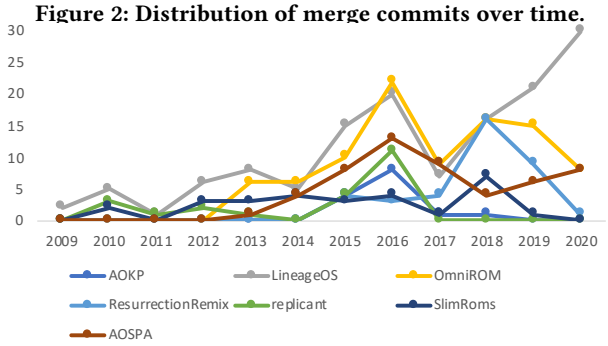

Figure 3: Merge commits with tags over time.

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 [1]. 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.

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

Table 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

was made in the upstream repository. Considering the example in Listing 1, the analysis computed the time lag between commit `80d8b6467ec4` and commit `37a24f52e6be`.

Results: Table 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 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 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.

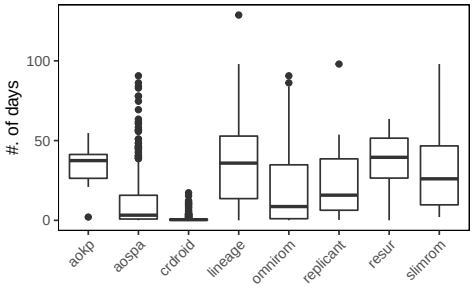

Figure 4: The distribution of merge time lag.

Table 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%)

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 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 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 1, 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 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**.

Table 5: Files affected by merge conflicts.

Downstream Project	*Files*						*Project*
Name	Java	aidl	cpp	xml	png	other	*Tot*
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

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?*

```
/**
  * <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 2: 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 3: 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 5 and 6 report the results of our analysis. Table 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 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 6 summarize the number of conflicts caused by the comments associated with classes, fields, constructors, methods, and inline comments or standalone comments not specific to any code. This part of Table 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 2 and 3 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 2. If downstream developers do

Table 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

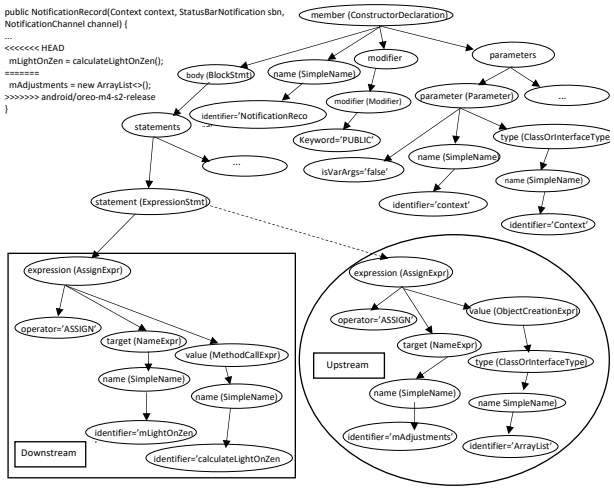

Figure 5: Sample AST.

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 [2] 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.

Fig. 5 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

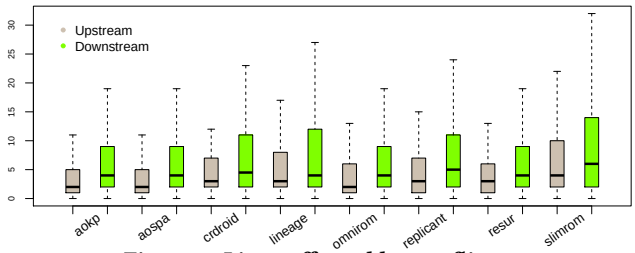

Figure 6: Lines affected by conflicts.

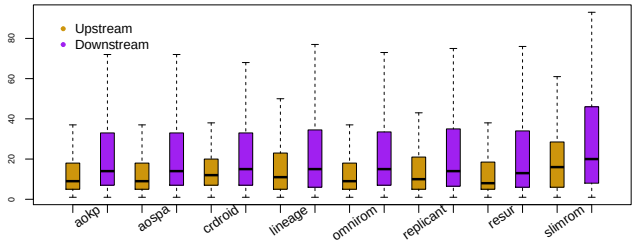

Figure 7: AST nodes affected by conflicts.

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

Results: Fig. 6 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. 7 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. 6, **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 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 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.

Table 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

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 [25]. 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), 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.

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 [34]. 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.

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 [4] or OnePlus [3]. 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.

7 RELATED WORK

There are other works that have studied merge conflicts [8, 10, 29, 31, 32, 38, 41] in divergent projects/branches. In this section, we describe related work closely related to ours.

Mens [31] 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. [29] 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. [12] and Accioly et al. [5] 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. [16] 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. [39] 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. [33] 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. [19] did an extensive systematic literature review of API evolution including Android and other Java-based APIs. Inspired by the approaches in existing literature [18, 35], 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.

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.

ACKNOWLEDGEMENTS

This work is supported by ARC Laureate Fellowship FL190100035, Discovery Early Career Researcher Award DE200100016, Discovery Project DP200100020.

REFERENCES

[1] 2021. *Codenames, Tags, and Build Numbers.* https://source.android.com/setup/start/build-numbers
[2] 2021. *Java parser.* https://javaparser.org/
[3] 2021. *OnePlus products.* https://en.wikipedia.org/wiki/OnePlus
[4] 2021. *Xiaomi products.* https://en.wikipedia.org/wiki/List_of_Xiaomi_products
[5] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018. 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*. 955–955.

[6] AOKP user bases 2013. AOKP user bases. 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-nightlies-available-now

[7] aosp 2021. About the Android Open Source Project. https://source.android.com

[8] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. 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*. 190–200.

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

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

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

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

[13] eproject 2021. /e/. https://e.foundation

[14] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China). Association for Computing Machinery, New York, NY, USA, 204–215.

[15] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2019. Understanding the Evolution of Android App Vulnerabilities. *IEEE Transactions on Reliability (TRel)* (2019).

[16] Gleiph Ghiotto, Leonardo Murta, Marcio Barros, and Andre Van Der Hoek. 2018. 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 (2018), 892–915.

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

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

[19] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A Systematic Review of API Evolution Literature. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–36.

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

[21] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*.

[22] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.

[23] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 254–264.

[24] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. CDA: Characterising Deprecated Android APIs. *Empirical Software Engineering (EMSE)* (2020).

[25] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2017. AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. *arXiv preprint arXiv:1709.05281* (2017).

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

[27] LineageOS 2021. LineageOS. https://lineageos.org

[28] LineageOS-GitHUb 2021. LineageOS GitHub. https://github.com/LineageOS

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

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

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

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

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

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

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

[36] Python-customized-scripts 2021. Customized Android: Dataset and Exploratory scripts. https://zenodo.org/record/6272071

[37] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. 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)*. IEEE, 288–298.

[38] Danhua Shao, Sarfraz Khurshid, and Dewayne E Perry. 2009. 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*. 291–292.

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

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

[41] Chungha Sung, Shuvendu K Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. 2020. 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*. 172–181.

[42] Daniel R Thomas, Alastair R Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. 2015. The lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface. In *Cambridge International Workshop on Security Protocols*. Springer, 126–138.

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

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

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

[46] Wikipedia 2021. List of custom Android distributions. https://en.wikipedia.org/wiki/List_of_custom_Android_distributions

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

[48] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. 2018. How Do Android Operating System Updates Impact Apps?. In *MobileSoft* (Gothenburg, Sweden). ACM, New York, NY, USA, 156–160.