# Dependency Facade: The Coupling and Conflicts between Android Framework and Its Customization

Wuxia Jin
*Xi'an Jiaotong University*
Xi'an, China
jinwuxia@mail.xjtu.edu.cn

Yitong Dai
*Xi'an Jiaotong University*
Xi'an, China
daiyitong0808@stu.xjtu.edu.cn

Jianguo Zheng
*Xi'an Jiaotong University*
Xi'an, China
Zz_Nut@stu.xjtu.edu.cn

Yu Qu
*University of California, Riverside*
California, the United States
yuq@ucr.edu

Ming Fan
*Xi'an Jiaotong University*
Xi'an, China
mingfan@mail.xjtu.edu.cn

Zhenyu Huang
*Honor Device Co., Ltd.*
Xi'an, China
huangzhenyu1@hihonor.com

Dezhi Huang
*Honor Device Co., Ltd.*
Xi'an, China
huangdezhi@hihonor.com

Ting Liu
*Xi'an Jiaotong University*
Xi'an, China
tingliu@mail.xjtu.edu.cn

***Abstract*—Mobile device vendors develop their customized Android OS (termed *downstream*) based on Google Android (termed *upstream*) to support new features. During daily independent development, the downstream also periodically merges changes of a new release from the upstream into its development branches, keeping in sync with the upstream. Due to a large number of commits to be merged, heavy code conflicts would be reported if auto-merge operations failed. Prior work has studied conflicts in this scenario. However, it is still unclear about the coupling between the downstream and the upstream (We term this coupling as the *dependency facade*), as well as how merge conflicts are related to this coupling. To address this issue, we first propose the *DepFCD* to reveal the *dependency facade* from three aspects, including *interface-level dependencies* that indicate a clear design boundary, *intrusion-level dependencies* which blur the boundary, and *dependency constraints* imposed by the upstream non-SDK restrictions. We then empirically investigate these three aspects (RQ1, RQ2, RQ3) and merge conflicts (RQ4) on the dependency facade. To support the study, we collect four open-source downstream projects and one industrial project, with 15 downstream and 15 corresponding upstream versions. Our study reveals interesting observations and suggests earlier mitigation of merge conflicts through a well-managed dependency facade. Our study will benefit the research about the coupling between upstream and downstream as well as the downstream maintenance practice.**

***Index Terms*—upstream, downstream, dependencies, merge conflict**

## I. INTRODUCTION

Mobile device vendors develop their customized Android OS based on Google Android, i.e., Android Open Source Project (AOSP) [16], to enhance Android OS functions, support new features, and add customized features. A customized Android, such as MiUI [35], MagicUI [30], and LineageOS [28], is usually termed the *downstream*, and the AOSP maintained by Google is termed the *upstream* [29]. Besides independent development, the downstream needs to periodically merge changes of a new release from the upstream into its development branches, keeping in sync with the upstream. Due to a large number of commits to be merged, heavy code conflicts (named *textual conflicts*) [20, 43, 46] would be reported by VCS (Version Control System) like Git [15] if auto-merge operations failed [29, 31].

Prior work has studied conflicts due to the merging of the upstream and downstream. The work of [31] investigated the refactorings made in the LineageOS and the upstream through commit history analyses. Liu et al. [29] empirically revealed the impacts of code conflicts on Android mobile applications. An industrial case study by Sung et al. [46] investigated merge-induced conflicts in Microsoft Edge development. However, it is still unclear how the downstream relies on the upstream and how the code conflicts are related to such a coupling.

This work will address this problem for Android. We first propose the DepFCD with an approach to reveal the dependency facade. Since a downstream project consists of the *native* code developed by upstream and the *extensive* code developed by downstream, DepFCD represents the downstream as a dependency graph. The *dependency facade* is a subgraph that links *native* and *extensive* nodes, manifested as the coupling between the downstream and upstream from three aspects. The first is the *interface-level dependency*, which indicates a clear design boundary of how the downstream relies on the upstream via *call*, *inherit*, *aggregate*, etc. The second is the *intrusion-level dependency*, which describes how the downstream directly modifies a portion of upstream code. Such intrusive operations would incur the *reverse dependency*, i.e., the upstream code depending on the downstream one. The third is the *dependency constraint* imposed by the upstream non-SDK restrictions [2]. Non-SDK restrictions proposed by Google indicate that some APIs in the upstream code might be unstable and frequently updated by the upstream. Following the Stable Dependency Principle (SDP) [33], we assume that the downstream code should depend upon the upstream code that is more stable than it is.

Next, we conduct an empirical study to investigate *interface-level* dependencies (**RQ1**), *intrusion-level* dependencies (**RQ2**), *dependency constraints* (**RQ3**), and merge conflicts (**RQ4**) on the dependency facade. We collected four open-source downstream projects and one industrial project for

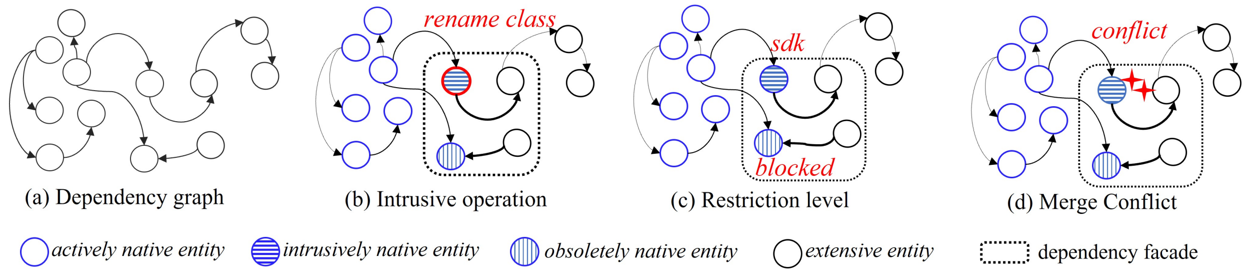


Fig. 1: The motivation of the DepFCD

study. Our dataset includes a total of 15 downstream versions, 15 corresponding upstream versions, 122M LoC, and 16M items of commit records in the revision history. Our study of the dependency facade reveals how the downstream code is intertwined with the upstream code, and how merge conflicts are related to the dependency facade characterization.

We reveal several major observations. 1) In general, interface-level dependencies from downstream to upstream dominate the dependency facade when compared to reverse dependencies that are induced by intrusion-level dependencies. 2) Reverse dependencies of *inherit*, *implement*, and *reflection* exist due to intrusive modifications to upstream code. Concretely, the downstream tends to conduct the fine-grained method block modification to upstream entities, followed by modifications of the class field, the import, and the modifier. 3) The non-SDK restriction level evolution in the downstream is basically consistent with that of the upstream. However, there exist both the loosening and enhancing of upstream restriction levels, which are modified by the downstream. 4) Merge conflicts are frequently caused by downstream changes of upstream entities, including modifications of method blocks, class fields, import statements, method parameters, and entity accessibility, as well as several violations of dependency constraints. Our results suggest that downstream development should be aware of the changes in their dependency facade and detect the violations of dependency constraints to ensure a clear design boundary, thus reducing potential merge conflicts. We also have reported the results to industrial experts responsible for the industrial project we studied. They confirmed exceptional observations and took our suggestions.

Our contributions include:

- We propose DepFCD to detect the dependency facade that couples downstream and upstream together.
- We conduct an empirical study to explore the dependency facade, including dependencies, constraints, and conflicts.
- We discuss study results, suggesting mitigation of merge conflicts through a well-managed dependency facade.
- We release our dataset[1] to facilitate the follow-up research in this direction.

[1] https://github.com/DepFCD/DepFCD

## II. Methodology

### A. The concept of DepFCD

By representing the downstream as a dependency graph with diverse attributes, we propose the DepFCD that defines the *dependency facade* as a sub-graph to characterize the coupling between the downstream and upstream. We depict Fig. 1 to explain the DepFCD model with key concepts. First, Fig. 1(a) represents a downstream project as a **dependency graph**. Each node denotes an entity $e$ in the source code such as *class*, *interface*, *method*, and *variable*; each directed edge $e_i \rightarrow e_j$ denotes a dependency from $e_i$ to $e_j$, such as *inherit* between classes, *aggregate* between classes, *call* between methods, etc.

We decompose the dependency graph of a downstream project to localize the **dependency facade**, a minimal sub-graph that couples a downstream code with the upstream one. As shown in Fig. 1(b), the graph decomposition produces three parts: *native* entities and their dependencies, *extensive* entities and their dependencies, and *dependency facade* which connects native entities and extensive ones. The **native entities** are originally created by upstream Android while might be further modified by the downstream. The **extensive entities** are created by the downstream and the upstream development is imperceptive of them. **Interface-level dependencies** on the dependency facade indicate how the downstream relies on the upstream through inherit, implement, aggregate, etc.

The *native* entities from the upstream project on a dependency facade might be intrusively modified by downstream development. As shown in Fig. 1(b), we classify *native* entities into three types: **actively native** entities which are native entities and keep unchanged in the downstream; **intrusively native** entities are originally created by the upstream while further modified by the downstream; **obsoletely native** entities are created by the upstream earlier versions while are "deprecated" by the current upstream version; however, such "obsolete" upstream entities are still reused by the downstream. Fig. 1(b) shows an *intrusively native* entity with a "rename class" operation. Those modifications to *intrusively native* entities by the downstream indicate a tighter coupling that makes the implementation logic of the downstream mixed together with the upstream. We term such coupling **intrusion-level dependencies** since it blurs the design boundary between the downstream and upstream, in contrast with *interface-level dependencies* that characterize a clear design boundary.

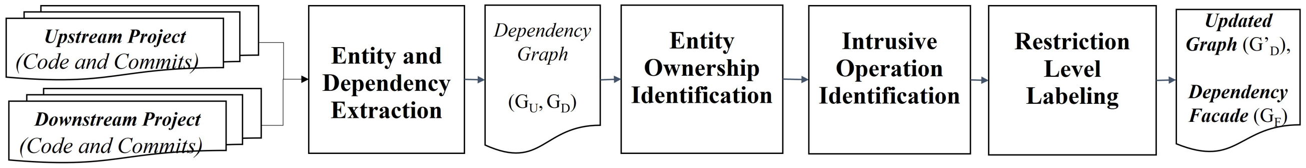


Fig. 2: The overview of the dependency facade detection

We assume that the upstream non-SDK restrictions [2] impose **dependency constraints** on the dependency facade–downstream entities should only depend upon the upstream entities that are more stable than they are, as indicated by the Stable Dependencies Principle (SDP) [33]. Non-SDK entities are internal and private API entities that are unstable. Depending on them would make downstream code frequently change to adapt to changes. The restriction level of non-SDK entities can be claimed as several kinds: *sdk* which can be publicly accessed; *blocked* which should not be accessed; *unsupported* which can be exploited by developers while would be changed in the future; and *max-target-"X"* which can be accessed in Android versions not later than the "X" version. Fig. 1(c) illustrates a *sdk* entity and a *blocked* entity.

Merge conflicts would be incurred when the downstream merges the update from the upstream (Fig. 1(e)). Our interest is the *textual conflicts* reported by CVS since their resolution is a high priority to make the updated OS quick to run, facilitating subsequent functional testing. Characterizing the relationship between these conflicts and the dependency facade features will suggest a potential direction to mitigate merge conflicts.

### B. The Detection of Dependency Facade

We design an approach to detect the dependency facade between a downstream project and the corresponding upstream project. Fig. 2 illustrates the approach overview, which will be introduced in following sections.

*1) Entity and Dependency Extraction:* This module employs a static code analysis tool named ENRE [23–25] to produce dependency graphs. $G_U$ is the upstream project and $G_D$ represents the downstream project. In the graph, nodes denote entities and edges denote the dependencies between them. The entities extracted by ENRE include *class*, *method*, *interface*, *annotations*, etc. The extracted dependencies include *inherit* between classes, *implement* from a class to an interface, *call* between methods, etc.

Each entity $e$ has several common properties: *id* uniquely denotes an entity; *qualifiedName* textually describes the name of an entity; *location* indicates its line number in the source file. Each dependency *dep* (from $e_i$ to $e_j$) also has several properties: *src* denotes the *id* of $e_i$; *dest* denotes the *id* of $e_j$; *value* records details like dependency kinds.

*2) Entity Ownership Identification:* The inputs of this module are the dependency graph ($G_D$) of the downstream project, the downstream project commits history ($Commits_D$), the commit history ($Commit_U$) of the corresponding upstream version, and the commit history ($Commit_O$) of the upstream older versions. Concretely, $Commit_O$ is a sub-set of upstream project's commits that meet two conditions: a) The commits do not exist in history checked out from the tagged upstream version that is merged by the downstream version; b) The commits exist in the whole upstream commit history.

This module identifies the ownership for each entity $e$ in $G_D$, recording the ownership label into entity property $e.ownership$. We categorize entity ownership into four types, i.e., *actively native*, *obsoletely native*, *intrusively native*, and *extensive*, as introduced in Section II-A.

The entity ownership identification is as follows. First, we leverage the command like *git blame* in Git [6] to obtain the commit set ($Commit_e$). $Commit_e$ is a set of commits where the modifications were made on $e$.

Given that $U_e = Commits_U \cap Commits_e$, $D_e = Commits_D \cap Commits_e$, and $O_e = Commits_O \cap Commits_e$, we determine the ownership of entity $e$ as follows:

*e.ownership*=*"actively native"* IF $U_e \neq \emptyset$, $D_e == \emptyset$, $O_e == \emptyset$;
*e.ownership* = *"obsoletely native"* IF ($U_e == \emptyset$, $D_e == \emptyset$, $O_e \neq \emptyset$) OR ($U_e \neq \emptyset$, $D_e == \emptyset$, $O_e \neq \emptyset$);
*e.ownership* = *"intrusively native"* IF ($U_e \neq \emptyset$, $D_e \neq \emptyset$, $O_e == \emptyset$) OR ($U_e == \emptyset$, $D_e \neq \emptyset$, $O_e \neq \emptyset$) OR ($U_e \neq \emptyset$, $D_e \neq \emptyset$, $O_e \neq \emptyset$);
*e.ownership* = *"extensive"* IF $U_e == \emptyset$, $D_e \neq \emptyset$, $O_e == \emptyset$

With entity ownership information, we detect the dependency facade $G_F =< V', D' >$. It is a sub-graph belonging to the downstream dependency graph $G_D =< V, D >$. Each edge $e_i \rightarrow e_j$ in $D'$ should satisfy one of the following conditions:

$e_i$*.ownership==*"extensive"* and $e_j$*.ownership.endswith("native")*;
$e_j$*.ownership==*"extensive"* and $e_i$*.ownership.endswith("native")*

*3) Intrusive Operation Identification:* This module identifies modification operations made by the downstream development to *intrusively native* entities. It combines the dependency graph analysis, the RefactoringMiner [40, 47, 48], and the *git blame* results in Section II-B2 to detect various modification operations.

Given a class or method entity $e_i$, $e_i$ is an *intrusively native* entity in the downstream dependency graph $G_D$ and its corresponding entity is $e_j$ in the upstream $G_U$. We compare $e_i$ with $e_j$ to determine whether the modifier (i.e., private, public, protected, final) keyword of upstream $e_j$ is newly added, modified, or removed. If such a change occurred, we label the downstream $e_i$*.operation* with that modification operation. For a file entity $e_i$ in downstream, we label it $e_i$*.operation* with the *"add the import"* if any *import* dependencies are newly introduced.

Similarly, we detect other operations by comparing $G_D$ and $G_U$. Considering that $e_i$ is a modified method entity, we set

$e_i$*.operation=“modify method parameter”* if the parameter list of $e_i$ is different from that of $e_j$. We set $e_i$*.operation=“modify method body”* if the code body is modified.

We also employ RefactoringMiner [40, 47, 48] to detect the refactoring operations that occurred on an *intrusively native* entity $e_i$ in the $G_D$. RefactoringMiner mines the commit history $Commits_D$ and uses the AST-based statement matching algorithm to determine refactoring candidates. Based on the results of RefactoringMiner, we assign $e_i$*.operation* with the refactoring operation types.

*4) Restriction Level Labeling:* This module will label the entity node with a *restrictlevel* attribute for the upstream and downstream dependency graph, $G_U$ and $G_D$. Following the official Android guidance [2], we first execute the command “`m out/soong/hiddenapi/hiddenapi-flags.csv`” under the source code of the downstream (or upstream) Android project. The execution outputs *hiddenapi-flags.csv* file that lists the restriction levels of non-SDK API entities.

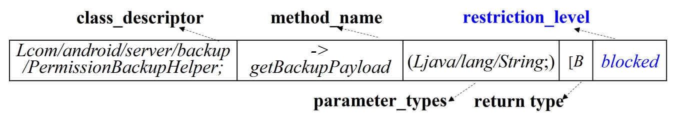


Fig. 3: An example in *hiddenapi-flags.csv*

Following the JNI specification [39], we decode the API signature in *hiddenapi-flags.csv*. Considering the Fig. 3 that indicates one row in this file, it can be split it into several attributes: *class_descriptor=“com.android.server.backup.PermissionBackupHelper”*, *method_name=“getBackupPayload”*, *parameter_types=“java.lang.String”*, *return_type=“ArrayList<byte>”*, and *restriction_level=“blocked”*. Using these attributes except the *restriction_level*, we search for corresponding entity $e$ that presents the same attributes in the dependency graph. Then we assign the restriction level to the right $e$, i.e., $e$*.restrictlevel=“blocked”*. We update the original dependency graph $G$ by labeling non-SDK entities with their restriction levels.

Finally, the approach outputs the dependency facade ($G_F$) and the updated $G'_D$, with diverse entity properties.

## III. Experimental Setup

### A. Research Questions

We will study four research questions to demystify the dependencies between downstream Android customizations and the Android framework on which the customizations are developed.

**RQ1. How do the downstream customizations rely on the upstream Android through interface-level dependencies?** With a clear design boundary indicated by interface-level dependencies, the downstream can easily evolve with upstream. We will study entity ownerships to expose the dependency facade and observe interface-level dependencies.

**RQ2. How do the downstream customizations rely on the upstream Android through intrusion-level dependencies?** Intrusion-level dependencies make fine-grained implementation logic of the downstream mixed together with the upstream. It would blur the design boundary. This study will investigate intrusion-level dependencies.

**RQ3. How do the downstream customizations adapt to the dependency constraint imposed by the upstream Android?** This study will explore how the dependency restriction evolution in downstream projects is consistent with that in upstream Android, and how downstream projects follow the dependency constraint on the dependency facade.

**RQ4. How do merge conflicts occur on the dependency facade between downstream customizations and the upstream Android?** We will understand how interface-level dependencies, intrusion-level dependencies, and dependency constraints would be related to merge conflicts.

### B. Subject and Version Collection

*1) Subject Collection:* We selected open-source Android-based projects from a curated list[1]. This list contains Android-based projects and is actively maintained. Our selection criteria are: 1) a project is managed with a version control system and thus we can collect its commit history; 2) the Android framework base of a project is continuously evolving with the AOSP and has a certain number of modifications since the year of 2020; 3) the project’s source code and its branches tagged with version information are available.

We also collected an industrial project anonymously named *IndustrialX* from an industrial well-known provider of smart devices. The project is also a customized extension of the AOSP. The industrial vendor has maintained its *IndustrialX* for 10 years. It takes the top 3 in domestic market share. In particular, its phone devices with AndroidOS serve for 40Million new users in the year of 2021.

Our study will focus on the Java code of the Android *framework base* of these subjects for two reasons. First, the *framework base* implements core functionalities via Java language. Second, downstream projects usually customize this part [31]. Finally, we collected five subjects for the investigation: four open-source projects and one industrial project. The *Project* column in Table I lists the collected projects.

*2) Project Version Collection:* We collected the project versions under investigation, including downstream project versions and their corresponding upstream Android versions on which they were built. The concrete steps are as follows.

For downstream projects, we selected the latest several branches to represent versions. Since our subjects use the Git to manage projects, we can easily obtain their branch versions. For a downstream version, we obtain the corresponding upstream version by analyzing merge commits in this downstream branch. Listing 1 illustrates a merge commit from the LineageOS-19.1. It reports the merge commit identifier `8acf7b` (line 1), the parent commits where `87f04d` is the commit ID in the LineageOS change history and `9cdf73` is the commit ID that will be merged into (line 2), the author making this change (line 3), the timestamp (line 4), and the commit log (line 5). We can identify that this commit merged the upstream changes (`9cdf73`) from *android-12.1.0_r7* into

TABLE I: The collected downstream project versions, the corresponding upstream versions, and their merge conflicts

| Project | Downstream Version | | | | Upstream (AOSP) Version | | | | Merge Commits | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Version | #File | #LoC | #Commit | Version | #File | #LoC | #Commit | #MerCmt | #CflCmt |
| AOSPA [4] | Sapphire | 27,539 | 4,575,525 | 674,953 | android12-gsi | 27,308 | 4,526,922 | 661,807 | 46 | 37 |
| | Ruby | 26,497 | 4,165,240 | 501,248 | android11-qpr1-d-release | 26,429 | 4,147,331 | 650,983 | 64 | 47 |
| | Quartz | 19,939 | 3,188,444 | 420,479 | android11-d2-release | 19,913 | 3,175,862 | 498,028 | 99 | 52 |
| CalyxOS [9] | android12 | 27,384 | 4,530,132 | 651,640 | android-12.0.0_r29 | 27,308 | 4,526,923 | 649,754 | 5 | 2 |
| | android11 | 26,699 | 4,169,261 | 482,229 | android-11.0.0_r46 | 26,463 | 4,155,454 | 497,949 | 10 | 1 |
| LineageOS [28] | LineageOS-19.1 | 27,685 | 4,576,171 | 663,065 | android-12.1.0_r7 | 27,469 | 4,558,550 | 659,793 | 3 | 1 |
| | LineageOS-18.1 | 26,713 | 4,190,090 | 499,427 | android-11.0.0_r46 | 22,534 | 3,669,603 | 505,590 | 17 | 8 |
| | LineageOS-17.1 | 22,951 | 3,711,094 | 426,543 | android-10.0.0_r41 | 22,455 | 3,652,993 | 571,185 | 28 | 12 |
| | LineageOS-16.0 | 2,0293 | 3,212,780 | 371,425 | android-9.0.0_r61 | 19,871 | 3,171,811 | 369,065 | 50 | 12 |
| OmniROM [38] | android-12.0 | 27,351 | 4,530,005 | 650,684 | android-12.0.0_r28 | 27,308 | 4,526,922 | 650,983 | 4 | 2 |
| | android-11 | 25,866 | 4,147,440 | 502,111 | android-11.0.0_r38 | 25,796 | 4,131,630 | 687,952 | 178 | 106 |
| | android-10 | 22,744 | 3,696,397 | 426,463 | android-10.0.0_r41 | 22,558 | 3,677,983 | 650,983 | 10 | 3 |
| | android-9 | 19,930 | 3,186,526 | 371,214 | android-9.0.0_r34 | 19,824 | 3,173,299 | 423,947 | 43 | 3 |
| IndustrialX | S | 35,546 | 5,866,790 | 661,243 | android-12.0.0_r10 | 27,161 | 4,511,991 | 648,567 | 2 | 2 |
| | R | 31,181 | 4,909,996 | 491,418 | android-11.0.0_r35 | 26,454 | 4,152,622 | 418,833 | 14 | 14 |
| **SUMMARY** | 15 versions | 388,318 | 62,655,891 | 7,794,142 | 15 versions | 368,851 | 59,759,896 | 8,545,419 | 573 | 302 |

LineageOS-19.1. Therefore, We regard the *android-12.1.0_r7* as an upstream version of LineageOS-19.1.

Listing 1: A merge commit log in the LineageOS-19.1

```
1 commit 8acf7b3981a55a3da70fd03d38b3645437b84d46
2 Merge: 87f04da03724 9cdf73f7cbed
3 Author: xxx <xxx@xxx.com>
4 Date:   Sat Jun 11 18:26:40 2022 +0300
5 Merge tag 'android-12.1.0_r7' into staging/lineage-19.1
       _merge-android-12.1.0_r7
```

The *Downstream Version* and *Upstream (AOSP) Version* columns in Table I list downstream project versions and the corresponding upstream AOSP versions we have collected. #File counts source files of *framework base* in a version; #LoC counts the lines of code in these files; #Commit counts the commit records during the version evolution. Our study will investigate this dataset, totally with 15 downstream versions, 15 upstream versions, 122M LoC, and 16M commit items.

## C. Merge Conflict Collection

We also collected merge conflicts of downstream projects when merging updates from the upstream Android framework. As explained in Listing 1 (Line 2), we obtained parent commit IDs of merge commits. Given a pair of parent commit IDs, we simulated merge operations locally. The merge simulation will generate conflict logs if conflicts happen. The *Merge Commits* column in Table I summarizes the merge and conflict information. #MerCmt counts the merge commits for the studied version. #CflCmt counts the merge commits that incurred code conflicts. Taking AOSPA-Sapphire as an example, 46 merge commits are collected and 37/46 commits trigger conflicting. *SUMMARY* indicates that 573 merge commits were collected from projects, where 52.7% merge commits are involved in code conflicts.

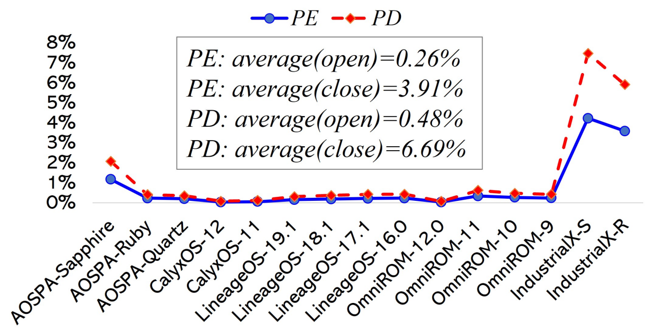


Fig. 4: The dependency facade size (*PE* and *PD*)

## IV. Experimental Results

### A. Interface-level Dependencies (RQ1)

*1) Setup:* We first computed $P_{act}$, $P_{ins}$, $P_{ext}$, and $P_{obs}$ to observe the proportion of entities with different ownerships in a downstream dependency graph $G'_D$. For instance, $P_{act} = \frac{|E_{act}|}{|E|}$ where $|E_{act}|$ and $|E|$ denote the number of *actively native* entities and all of the entities.

Next, we measured *PE* and *PD* to calculate the size of a dependency facade $G_F$. $PE = \frac{|E_F|}{|E|}$ denotes the proportion of entities $E_F$ in $G_F$ to all of the entities $E$ in $G'_D$; $PD = \frac{|D_F|}{|D|}$ denotes the proportion of dependencies ($D_F$) in $G_F$ to all of the dependencies $D$ in $G_D$.

We then analyzed interface-level dependencies *downstream* $\rightarrow$ *upstream* (i.e., $D \rightarrow U$) located on the dependency facade. We also pinpointed the downstream and upstream entities which frequently participate in interface-level dependencies.

*2) Results:* Table III lists the entity distribution in terms of their ownership, averaged on open-source versions and industrial versions respectively. It shows the **Observation 1**. Concretely, $P_{ins}$ and $P_{ext}$ for industrial projects are larger than those for open-source projects. That is, industrial projects are prone to extending more functionalities and making more changes to the upstream.

TABLE II: $D \rightarrow U$ dependencies

| Project | AGG | IMP | INH | CAL | OVR | ANN | REF |
|---|---|---|---|---|---|---|---|
| AOSPA-Sapphire | 608 | 75 | 78 | 9,479 | 107 | 223 | 259 |
| AOSPA-Ruby | 69 | 25 | 10 | 1,482 | 11 | 22 | 19 |
| AOSPA-Quartz | 14 | 9 | 3 | 1,012 | 5 | 18 | 8 |
| CalyxOS-12 | 25 | 3 | 13 | 190 | 68 | 2 | 10 |
| CalyxOS-11 | 30 | 2 | 14 | 516 | 55 | 0 | 11 |
| LineageOS-19.1 | 67 | 33 | 30 | 1,016 | 114 | 54 | 28 |
| LineageOS-18.1 | 74 | 26 | 36 | 1,407 | 136 | 6 | 27 |
| LineageOS-17.1 | 101 | 24 | 33 | 1,499 | 123 | 25 | 44 |
| LineageOS-16.0 | 72 | 17 | 30 | 1,311 | 107 | 2 | 29 |
| OmniROM-12 | 13 | 0 | 14 | 289 | 39 | 1 | 1 |
| OmniROM-11 | 155 | 32 | 40 | 2,694 | 94 | 22 | 28 |
| OmniROM-10 | 100 | 11 | 42 | 2,149 | 108 | 17 | 22 |
| OmniROM-9 | 81 | 6 | 32 | 1,774 | 75 | 3 | 10 |
| IndustrialX-S | 4,535 | 456 | 965 | 86,573 | 1,298 | 148 | 538 |
| IndustrialX-R | 2,382 | 207 | 534 | 60,166 | 1,035 | 273 | 229 |

NOTE: AGG=Aggregate, IMP=Implement, INH=Inherit, CAL=Call, OVR=Override, ANN=Annotate, REF=Reflect

TABLE III: The entity ownership distribution

| Project | $P_{act}$ | $P_{obs}$ | $P_{ins}$ | $P_{ext}$ |
|---|---|---|---|---|
| AVERAGE (open) | 99.22% | 0.07% | 0.24% | 0.47% |
| AVERAGE (close) | 78.62% | 0.01% | 1.66% | 19.70% |

Fig. 4 shows the facade size measurements, *PE* and *PD*. Taking AOSPA-Sapphire as an example, 1.17% entities and 2.07% dependencies are located at the dependency facade, which couples AOSPA-Sapphire with its corresponding upstream Android (android12-gsi). The values labeled with Fig. 4 indicate the *PE* and *PD* values averaged on the open-source versions and industrial versions. It shows that *PD* and *PE* in the IndustrialX are $\frac{3.91\%-0.26\%}{0.26\%} = 14$ times and $\frac{6.69\%-0.48\%}{0.48\%} = 13$ times larger than those of open-source downstream projects (**Observation 2**).

Table II shows interface-level dependencies in the dependency facade. The number of *Inherit* and *Implement* dependencies is smaller than that of *Aggregate* (**Observation 3**). The *Call* accounts for the most. The *Override* dependencies between methods exist due to the *Inherit* or *Implement* dependencies. *Annotate* dependencies indicate that *extensive* classes are decorated with *native* annotation entities.

Table II shows the presence of *reflect* dependencies (**Observation 4**). Java reflection allows developers to call the methods and properties of any object at runtime, even if these class members are *private* [27]. For example in OminiROM-11, downstream `com.android.settingslib.wifi.AccessPoint.isSuiteBSupported` calls the upstream `android.net.wifi.WifiManager.getCapabilities()` through reflecting `android.net.wifi.WifiManager`. This method obtains a textual description of the driver at runtime, filtering the networks incompatible with the running driver.

Besides, we pinpointed the representative downstream packages which frequently employ the upstream and the upstream entities which are frequently depended upon (**Observation 5**).

**Observations in RQ1**

1) Among code entities in a downstream project, actively native entities account for the most, followed by *extensive* entities.
2) Industrial project exposes 13.5 times ($(13 + 14)/2$) larger dependency facade than that of open-source projects, indicating more design decisions to interact with the upstream.
3) Downstream development considers *aggregate* in priority to employ upstream entities rather than *implementation* and *inherit*.
4) The Downstream development employs Java reflection to access the properties of upstream entities at run-time.
5) The downstream tightly depends on the upstream to customize their functions related to the hardware, package manager, and system UI. The log, view, hardware, and content functions of upstream Android are commonly employed by the downstream.

## B. Intrusion-level Dependencies (RQ2)

*1) Setup:* We first observed diverse intrusive operations made by downstream projects to the upstream. We then analyzed reverse dependencies (the dependencies from upstream to downstream entities), due to intrusive modifications. We also pinpointed *hotspot* files, i.e., the *intrusively native* files which continue to be modified by downstream projects.

TABLE IV: A summary of intrusive operations

| Project | MDF | ANN | IPT | PCL | ITF | ICL | CFD | MBL | OTH |
|---|---|---|---|---|---|---|---|---|---|
| AOSPA-Sapphire | 485 | 62 | 1,782 | 23 | 38 | 67 | 2,948 | 9,485 | 468 |
| AOSPA-Ruby | 51 | 5 | 141 | 0 | 0 | 11 | 999 | 2,564 | 26 |
| AOSPA-Quartz | 35 | 1 | 108 | 0 | 0 | 8 | 687 | 983 | 13 |
| CalyxOS-12 | 1 | 0 | 38 | 0 | 2 | 1 | 77 | 343 | 7 |
| CalyxOS-11 | 1 | 0 | 57 | 0 | 0 | 3 | 218 | 340 | 8 |
| LineageOS-19.1 | 46 | 0 | 181 | 1 | 12 | 20 | 488 | 1,084 | 10 |
| LineageOS-18.1 | 50 | 0 | 197 | 1 | 12 | 17 | 653 | 1,022 | 22 |
| LineageOS-17.1 | 63 | 19 | 203 | 1 | 9 | 15 | 633 | 1,277 | 27 |
| LineageOS-16.0 | 58 | 0 | 338 | 2 | 8 | 13 | 573 | 1,156 | 27 |
| OmniROM-12 | 4 | 0 | 65 | 0 | 6 | 6 | 128 | 278 | 6 |
| OmniROM-11 | 61 | 3 | 309 | 0 | 11 | 18 | 1,105 | 2,418 | 40 |
| OmniROM-10 | 51 | 0 | 260 | 1 | 10 | 18 | 843 | 1,033 | 24 |
| OmniROM-9 | 79 | 0 | 157 | 0 | 7 | 13 | 674 | 828 | 15 |
| IndustrialX-S | 759 | 334 | 3,753 | 36 | 41 | 70 | 6,463 | 18,121 | 112 |
| IndustrialX-R | 725 | 20 | 3,178 | 38 | 39 | 57 | 5,405 | 14,067 | 158 |
| AVERAGE (open) | 75 | 6 | 295 | 2 | 8 | 16 | 771 | 1,754 | 53 |
| AVERAGE (close) | 742 | 177 | 3,465 | 37 | 40 | 63 | 5,934 | 16,094 | 135 |

NOTE: MDF=Modifier: ①*modify the accessibility*, ②*modify the final*, ANN=Annotation: ③*modify the annotation*, IPT=Import: ④*modify the import*, PCL=Parent Class: ⑤*modify the inherited parent class*, ITF=Interface: ⑥*modify the implemented interface*, ICL=Inner Class: ⑦*modify the inner class*, CFD=Class Field: ⑧*modify class field*, MBL=Method Block: ⑨*modify method parameter or return*, ⑩*modify local variable*, ⑪*modify method body*, ⑫*modify method annotation or parameter annotation*, OTH=Others: ⑬*rename or move class*, ⑭*rename or move method*, ⑮*extract method*.

*2) Results:* Table IV lists intrusive modifications conducted by downstream projects on *intrusively native* entities. Considering *AOSPA-Sapphire*, 567 intrusive operations are *Modifier* including *modify the accessibility* and *modify the final*. The **Observation 1** is consistent in the studied projects.

First, among the method block modifications, *modify method body* takes the most cases. One frequent change is that downstream development defines new condition branches inside the method. Second, regarding *modify the accessibility/final*, we found that the downstream development tends to promote accessibility and remove the final keyword. Third, *import* statement changes will impact the original dependencies built by upstream Android. Moreover, *rename/move class* and *rename/move/extract method* refactoring operations is less likely to occur than other modifications.

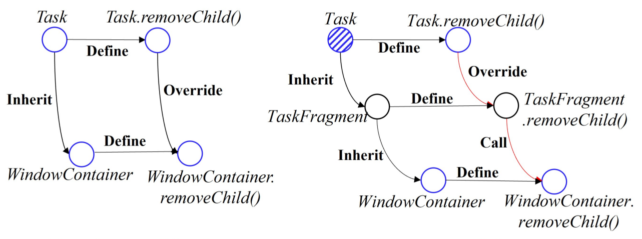
TABLE V: $D \leftarrow U$ dependencies due to intrusive operations

| Project | AGG | IMP | INH | CAL | OVR | ANN | REF |
|---|---|---|---|---|---|---|---|
| AOSPA-Sapphire | 246 | 20 | 2 | 3,554 | 28 | 8 | 27 |
| AOSPA-Ruby | 41 | 0 | 0 | 451 | 0 | 0 | 5 |
| AOSPA-Quartz | 34 | 0 | 0 | 392 | 0 | 0 | 0 |
| CalyxOS-12 | 0 | 0 | 0 | 143 | 0 | 0 | 0 |
| CalyxOS-11 | 2 | 0 | 0 | 67 | 0 | 0 | 0 |
| LineageOS-19.1 | 14 | 0 | 0 | 374 | 0 | 0 | 2 |
| LineageOS-18.1 | 16 | 0 | 0 | 354 | 0 | 0 | 2 |
| LineageOS-17.1 | 10 | 0 | 0 | 353 | 0 | 0 | 3 |
| LineageOS-16.0 | 11 | 0 | 0 | 426 | 0 | 0 | 2 |
| OmniROM-12 | 2 | 5 | 0 | 90 | 0 | 0 | 12 |
| OmniROM-11 | 47 | 11 | 0 | 651 | 0 | 0 | 26 |
| OmniROM-10 | 15 | 4 | 0 | 425 | 0 | 0 | 7 |
| OmniROM-9 | 15 | 1 | 0 | 351 | 0 | 0 | 0 |
| IndustrialX-S | 305 | 42 | 38 | 7,832 | 33 | 14 | 15 |
| IndustrialX-R | 280 | 42 | 40 | 7,372 | 46 | 0 | 9 |

Table V summarizes the reverse dependencies from upstream entities to downstream entities, presenting the **Observation 2**. We also found that fine-grained method calls tend to occur within an individual upstream class via the *modify method block*.

Table V also indicates the **Observation 3**. Concretely, Most of $D \leftarrow U$ *Aggregation*, *Inherit*, *Implement*, *Override*, and *Annotate* only appear in AOSPA-Sapphire and IndustrialX. We found a frequent observation in reverse dependencies of *Inherit* and *Implement*: the downstream development employs the *modify the import* to introduce a new base-class (or interface) for the *native* class. Sometimes, it even directly replaces the base-class (or an interface) of the *native* class with *extensive* one. We use an *Inherit* case in the AOSPA-Sapphire to give a detailed explanation here. Fig. 5(a) shows that `vm.Task` originally inherits `vm.WindowContainer` in upstream Android, and a method of the former class overrides a method of the latter. Afterward, the downstream code defines and imports an extensive class `vm.TaskFragment`, as shown in Fig. 5(b). AOSPA-Sapphire then changes original base-class of `vm.Task` into `vm.TaskFragment`, which continues to inherit the previous `vm.WindowContainer`. Such a change deepens the design hierarchy of the upstream. Consequently, the downstream would have to continue modifying other upstream entities influenced by this base-class.

The *Reflect* dependencies from upstream entities to downstream entities exist in Table V. We profiled the reflection usage and obtained the **Observation 4**. In Fig. 6(a), upstream method `com.android.systemui.qs.QSPanel.onAttachedToWindow` reflects a class `com.android.systemui.omni.OmniSettingsService` in OmniROM-11 to its `addIntObserver()`. This



(a) Before modification (b) After modification

Fig. 5: *Inherit* reverse dependency due to AOSPA-Sapphire changes into upstream *com.android.serverpackage.wm*

operation enables OmniRom to implement shortcut settings for specific functionality. Fig. 6(b) is another case of the reflection usage for a mock testing [45]in AOSPA-Sapphire.

```
// OmniROM-11, com.android.systemui.qs.QSPanel
protected void onAttachedToWindow() {
    super.onAttachedToWindow();
   Dependency.get(OmniSettingsService.class).
          addIntObserver(this, QS_SHOW_SECURITY);
   Dependency.get(OmniSettingsService.class).
          addIntObserver(this, Settings.System.
          OMNI_QS_SHOW_MEDIA_DIVIDER);
}
```

(a) Using reflection for special functionality realization

```
// AOSPA-Sapphire, com.android.systemui.media.dialog
public class MediaOutputControllerTest extends
       SysuiTestCase {
    private final SystemUIDialogManager mDialogManager =
          mock(SystemUIDialogManager.class);}
```

(b) Using reflection for mock testing

Fig. 6: The cases for $D \leftarrow U$ *Reflect* dependencies

In addition, we pinpointed the *hotspot* upstream entities that are commonly modified by all of the downstream projects and revealed the **Observation 5**.

Observations in RQ2

1) Downstream projects tend to conduct the *method block modification* to the upstream, followed by the modifications of class fields, import statements, and modifiers. These modifications lead to reverse dependencies, i.e., upstream entities depend on downstream ones.
2) The combination of Table II and Table V indicates that dependencies from the downstream to the upstream dominate the dependency facade although reverse dependencies exist.
3) *Inherit* and *Implement* reverse dependencies rarely happen except a few cases. Their usage deepened the dependency hierarchy or even broke the design hierarchy originally created by the upstream.
4) Reflection dependencies of $D \leftarrow U$ are due to mock testing and business functionality customization.
5) `com.android.server.pm.PackageManagerService` and `android.provider.Settings` are commonly modified by the downstream to customize the settings and APK management.

## C. Dependency Constraints (RQ3)

*1) Setup:* According to II-B4, the restriction level labeling takes “hiddenapi-flags.csv” as the input that records the restriction information. An accurate export of such files requires the

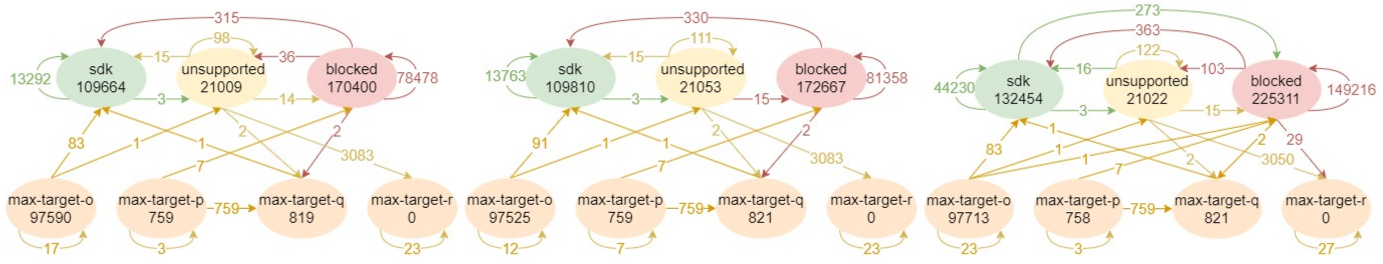

(a) Android-11 → Android-12 (b) LineageOS-18.1 → LineageOS-19.1 (c) IndustrialX-R → Industrial-S

Fig. 7: The state transfer graphs for restriction level changes from one version to another version

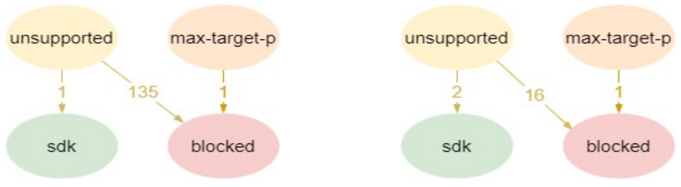

(a) Android-11.0.0_r35 → IndustrialX-R (b) Android-12.0.0_r10 → IndustrialX-S

Fig. 8: Restriction level changes made by the IndustrialX

whole Android repositories besides the Android framework to be available. We only successfully cloned the entire repositories for the LineageOS and Android AOSP and failed to fetch others due to the huge-size repositories. The industry of IndustrialX also provided locally stored repositories for us. Therefore, our RQ2 study used the LineageOS and IndustrialX projects. Moreover, LineageOS-19.1 and LineageOS-18.1 are developed based on Android-12 (i.e., S version) and Android-11 (i.e., R version), which are consistent with the upstream versions of IndustrialX-S and IndustrialX-R. Finally, our study investigated the LineageOS (19.1 and 18.1), IndustrialX (S and R), and the corresponding two upstream versions.

We first employed a *state transform graph* to model restriction level changes of API entities, as shown in Fig. 7. Each node in the graph denotes a restriction level (i.e., *sdk*, *block*, *unsupported*, *max-target-"X"*), and each edge denotes the changes from one restriction level to another. Based on this model, we observed the changes from the upstream Android-11 to upstream Android-12, from the LineageOS-18.1 to LineageOS-19.1, and from the IndustrialX-R to IndustrialX-S, respectively. We further examined restriction level changes of upstream Android API entities made by downstream versions.

Besides, we computed the *PS* that measures to what extent the downstream API entities call the *blocked*, *unsupported*, and *sdk* upstream entities on the dependency facade. Among those entities, we further assessed the *PI*, i.e., the proportion of *intrusively native* upstream non-SDK entities that were modified by downstream projects.

*2) Results:* Fig. 7 illustrates three state transform graphs to depict restriction level changes of API entities from Android-11 to Android-12 (a), from LineageOS-18.1 to LineageOS-19.1 (b), from IndustrialX-R to IndustrialX-S (c). In Fig. 7(a), each node denotes a restriction level state; the number inside a node denotes the number of API entities whose restriction level keeps unchanged from Android-11 to Android-12; the number labeled in an edge from a node $restrictlevel_i$ to another node $restrictlevel_j$ counts the API entities whose state is $restrictlevel_i$ in Android-11 while is changed into $restrictlevel_j$ in Android-12; especially, the number in the edge (loop) from a node (i.e., $restrictlevel_i$) to itself counts the newly added API entities with $restrictlevel_i$ in Android-12.

Several major observations (**Observation 1**) in Fig. 7 are as follows. First, considering the loops in Fig. 7(a), 91,911 (13,292+98+78,478+17+3+23=91,911) API entities have been newly added; *blocked* entities account for $78,478/91,911 = 85.4\%$; *sdk* entities take $13,292/91,911 = 14.5\%$ . Similarly, the proportion values are $85.5\%$ and $14.4\%$ for LineageOS, and $77.1\%$ and $22.8\%$ for IndustrialX. Second, the evolution from *blocked* to *sdk* exists in three sub-figures.This change makes the previously inaccessible APIs become available in the new version. About 3000 *unsupported* API entities in the older version have evolved into *max-target-o* ones. We found an exceptional observation (**Observation 2**) in the IndustrialX: In Fig. 7(c), the accessibility of 273 API entities were radically downgraded, i.e., from *sdk* into *blocked*.

Fig. 8 depicts the restriction level changes to upstream API entities made by the downstream. There is no change to upstream API entities in LineageOS-18.1 and LineageOS-19.1. Therefore, Fig. 8 only shows the results in IndustrialX-R and IndutrialX-S, revealing the **Observation 3**. An explanation in Fig. 8 is similar to Fig. 7. The only difference is that Fig. 7 has no loops and numbers inside nodes.

Fig. 8(a) demonstrates that the restriction level of 137 *native* entities in upstream Android-12.0.0_r10 has been updated by IndustrialX-R. $135/137 = 98.5\%$ changes are the degradation from *unsupported* to *blocked*, making entities completely inaccessible. The value in Fig. 8(b) is $84.2\%$ for IndustrialX-S version. Similarly, one degradation change occurs from *max-target-p* to *blocked* in Fig. 8(a) and (b). The restriction level loosening also exists– 3 upstream API entities were changed from *unsupported* to *sdk* by IndustrialX in Figure 8.

*PS* and *PI* measurements in Table VI indicate upstream non-SDK API usage on the dependency facade (see the **Observation 4**). For instance, among all of the upstream non-SDK APIs depended by LineageOS-19.1, 86.7% entities are *sdk*, 9.4% entities are *unsupported*, and 3.9% entities are *blocked*. Besides, among *blocked* upstream entities depended

TABLE VI: Upstream non-SDK API usage

| Project | *blocked* | | *unsupported* | | *sdk* | |
|---|---|---|---|---|---|---|
| | *PS* | *PI* | *PS* | *PI* | *PS* | *PI* |
| LineageOS-19.1 | 3.9% | 25.0% | 9.4% | 1.5% | 86.7% | 0.6% |
| LineageOS-18.1 | 2.1% | 21.7% | 10.1% | 5.3% | 87.8% | 0.0% |
| IndustrialX-S | 3.2% | 14.0% | 18.0% | 6.8% | 78.8% | 4.4% |
| IndustrialX-R | 2.9% | 12.3% | 22.2% | 4.9% | 74.9% | 4.5% |

by LineageOS-19.1, 25.0% of them are *intrusively native* blocked APIs which were further modified by LineageOS-19.1. The downstream versions exhibit a consistent observation.

As illustrated in the **Observation 5**, we pinpointed several *blocked* APIs that are frequently depended upon by the downstream, which would violate the SDP. The *blocked* API `android.app.ActivityTaskManager. getService` was frequently accessed by both LineageOS and IndustrialX. Downstream versions employed this *blocked* API to acquire objects of `ActivityManagerService` to manage the activity. LineageOS also frequently accesses *blocked* APIs of `com.android.internal.widget.LockPatternUtils` to customize the lock pattern services. IndustrialX frequently uses *blocked* APIs of `com.android.internal.util` to acquire handler and manage messages in the state machine.

Observations in RQ3

1) The restriction level evolution in the downstream is basically consistent with that of the upstream. Most of the newly added APIs are *blocked*. There exists changes of *unsupported→max-target-"X"* and *blocked→sdk*.
2) The change of *sdk→blocked* uniquely exists in the industrial project, making previously accessible APIs unavailable radically.
3) No changes have been made by LineageOS to the upstream. On the contrary, both restriction enhancement of *unsupported→blocked* and restriction loosening of *unsupported→sdk* exist in IndustrialX.
4) The downstream tends to rely on *sdk* APIs as well as a small portion of *blocked* APIs. A larger portion of upstream *block* APIs on the dependency facade are further modified by the downstream.
5) The *blocked* APIs of `android.app.ActivityTaskManager .getService`, `com.android. internal.widget.LockPatternUtils`, and `com.android. internal.util.StateMachine` are frequently depended on by downstream projects.

## D. Merge Conflict (RQ4)

*1) Setup:* We collected 573 merge commits in total that downstream projects merge updates from upstream Android. As aforementioned in Table I, 52.7% (302/573) merge commits trigger conflicts. Focusing on these conflict commits, we counted Java source files (*#CflFil*), lines of code (*#CflLoc*), the averaged *#CflFil* on all of the conflict commits (*#aFIL*), and the averaged *#CflLoC* (*#aLOC*).

The column $Cfl_F$ in Table VII summarizes conflict files on the dependency facade. Based on those conflict files on the dependency facade, we randomly sampled 300 conflict blocks in them for a manual analysis. The number of sampled cases in each project is listed in column *#Sample* of Table VII. We profiled the samples to understand how the dependency facade would be involved in merge conflicts.

*2) Results:* Table VIII summarizes characterization results for conflict blocks on the dependency facade. Taking the first row in Table VIII as an example, *Method block* denotes that the conflicts are reported inside method blocks. *Count=132* denotes 132 conflict cases occurring in method blocks. The third column lists cases of different downstream modifications that result in merge conflicts. For instance, among 132 conflict cases occurring inside method blocks, 45 cases are due to downstream version modifying or adding condition branch inside the method body; 43 cases are related to call site changes; 24 cases are about method local variable changes; 10 cases are due to log string update; the remaining 10 cases are in other situations. Last column describes the features of the conflicts on the dependency facade: the conflicts are involved in *Call* dependencies from downstream entities to upstream ones; the conflicts caused by the side of downstream modifications are related to intrusive operations, including *modify local variable*⑩ and *modify method body*⑪. These numbers of intrusive operations refer to those in Table IV.

TABLE VII: The conflict information when downstream projects merge updates from the upstream Android

| Project | #CflFil | #CflLoc | #aFIL | #aLoc | #Cfl$_F$ | #Sample |
|---|---|---|---|---|---|---|
| AOSPA-Sapphire | 387 | 8,102 | 10.5 | 20.9 | 305(78.8%) | 28 |
| AOSPA-Ruby | 381 | 9,596 | 8.1 | 25.2 | 299(78.5%) | 26 |
| AOSPA-Quartz | 272 | 9,022 | 5.2 | 33.2 | 229(84.2%) | 33 |
| CalyxOS-12 | 21 | 307 | 10.5 | 14.6 | 2(9.5%) | 2 |
| CalyxOS-11 | 1 | 1 | 1.0 | 1.0 | 0(0.0%) | 0 |
| LineageOS-19.1 | 2 | 10 | 2.0 | 5.0 | 2(100.0%) | 5 |
| LineageOS-18.1 | 20 | 135 | 2.5 | 6.8 | 12(60.0%) | 12 |
| LineageOS-17.1 | 53 | 686 | 4.4 | 12.9 | 30(56.6%) | 55 |
| LineageOS-16.0 | 26 | 156 | 2.2 | 6.0 | 19(73.1%) | 7 |
| OmniROM-12 | 10 | 121 | 5.0 | 12.1 | 2(20.0%) | 2 |
| OmniROM-11 | 734 | 19,471 | 6.9 | 2.5 | 550(74.9%) | 61 |
| OmniROM-10 | 38 | 466 | 12.7 | 12.3 | 30(78.9%) | 12 |
| OmniROM-9 | 11 | 228 | 3.7 | 20.7 | 8(72.7%) | 6 |
| IndustrialX-S | 537 | 92,503 | 268.5 | 172.3 | 513(95.5%) | 51 |
| IndustrialX-R | 646 | 89,351 | 46.1 | 138.3 | 561(86.8%) | 0 |
| SUMMARY | 3,139 | 230,155 | 10.4 | 73.3 | 2,562(81.6%) | 300 |

In general, Table VIII shows the frequent downstream modifications that cause merge conflicts, revealing the **Observation 1**. It can be seen that 16 conflicts have an indirect relationship with downstream medications. They were reported around modification contexts instead of the modification occurrence. 49 conflicts are due to merging upstream modifications of the AOSP into the *actively native* entities, irrelevant to downstream modifications.

The last column in Table VIII indicates the **Observation 2**. A possible reason for the observation is that such *rename* or *move* refactoring operations would induce indirect conflicts (see the last row) reported around them. There is one conflict due to the refactoring of *extract method*⑮ among sampled cases. For example, LineageOS (Commit:*a5464e*) extracts a new method `updateIsKeyguard(boolen force)` from the body of upstream `updateIsKeyguard()`. Then this newly extracted method is called by the original `updateIsKeyguard()` .

Table VIII also indicates that method *call* dependencies appear mostly in conflicts, which is expected due to its fine granularity. More interestingly, intrusive modifications of *modify the inherited class*⑤ and *modify the implemented interface*⑥ indeed lead to conflicts (**Observation 3**). As discussed in RQ2 results, such changes break the design hierarchy, potentially resulting in consequent changes of the upstream to adapt them and thus incurring conflicts. We have examined such a case of Fig. 5 in RQ2 results: the parent class modification into an upstream *Task* leads to a reverse dependency (i.e,, *native* class inherits the *extensive* class), along with code conflict inside this class (merge commit *#26017c* in AOSPA ).

We can also see the **Observation 4**. As shown in *Class Field* row, 8 conflicts are in the cases where the downstream modifies or adds class static constants which are annotated with non-SDK restriction. Another conflict case (*Annotation* row) is also related to the dependency constraint. For example, IndustrialX development removes the non-SDK restriction level annotation `@UnsupportedAppUsage` of an upstream entity `SOURCE_CODEC_TYPE_MAX`. Consequently, this entity becomes from *unsupported* to *sdk*. Since *unsupported* APIs are unstable and would be changed by the upstream, merging with the upstream changes leads to this conflict.

Observations in RQ4

1) Merge conflicts are frequently caused by downstream changes of upstream entities, including modifications of *method blocks, class fields, import statement, method parameters, entity accessibility* and the new method definition inside upstream classes.
2) The *rename/move class or method* rarely leads to direct merge conflicts, while perhaps causing indirect conflicts around change contexts.
3) Reverse dependencies (i.e., *Call, Inherit, Implement*) due to intrusive changes are involved in code conflicts.
4) Merge conflicts also occurred in non-SDK API changes, which are related to dependency constraints.

## V. Discussion

Besides interface-level dependencies from the downstream to the upstream, reverse dependencies caused by intrusive modifications also exist on the dependency facade. This incurs a high-level *cyclic dependency* between the downstream and upstream. Although reverse dependencies are inevitable, these intertwined dependencies would blur dependency boundaries between the upstream and downstream. Consequently, it would incur heavy merge conflicts when the downstream merges with the upstream updates (see conflict analysis in RQ4). Industrial experts (from the IndustrialX) confirmed our points. **We suggest the downstream development should monitor the dependency facade changes to be a cleaner design, facilitating the potential prevention of merge conflicts.**

Our results indicate that non-SDK APIs were originally inaccessible *blocked* in the upstream AOSP while become available *sdk* by IndustrialX. We have reported the exceptional APIs to project experts. They checked and explained that the restriction loosening is due to the mistake made by developers. They agreed that such a change to the upstream should be prohibited since it violates the dependency constraints of upstream Android. This violation would impact the evolution of downstream projects, potentially leading to merging conflicts. Our conflict cases also demonstrate such conflicts. **To address potential conflict risks earlier, we recommend detecting dependency restriction violations during the upstream and downstream evolution.**

## VI. Threats to Validity

First, our DepFCD employs the ENRE [23–25] for entity and dependency extraction. ENRE supports extracting implicit dependencies caused by dynamic features, which other tools fail to identify [23, 25]. Second, the accuracy of *Entity Ownership Identification* and *Intrusive Operation Identification* of our DepFCD would impact the study of RQ1 and RQ2. To reduce threats, we combined the *git blame* command [6] and advanced *RefactoringMiner* [40, 47, 48] for an accurate analysis of commit history. *git blame* [6] traces code modifications and *RefactoringMiner* identifies refactoring operations involved in modifications. *RefactoringMiner* has been widely adopted in diverse work [22, 26, 42, 49].

Our RQ3 employed the *Restriction Level Labeling* of DepFCD to assign the non-SDK restriction levels documented in *hiddenapi-flags.csv* into the corresponding entities. To mitigate possible threats, we fetched and compiled the entire huge-scale project repositories to generate accurate *hiddenapi-flags.csv* files.

Our RQ4 conducted a manual study on the code conflicts on the dependency facade. To mitigate the possible subjectivity, the two authors of this work independently analyzed the conflict cases and reached consistent results. Moreover, we reported the results on IndustrialX to its developers. They confirmed our observations on these cases, as discussed in Section V. We will analyze more conflict cases. Moreover, besides the "merge" operation, "rebase" events [21] and "cherry-pick" [7] commits may also lead to conflicts. Our future work will explore this point.

At last, our work studied five customized Android projects. One potential threat is that different projects would present inconsistent observations. To reduce this threat, we collected multiple versions for each project. We also will study the dependency facade and merge conflicts in other ecosystems like Linux.

## VII. Related Work

**Dependency-based Software Understanding.** Much work has employed code dependencies to understand the software design. The work of [41] assumed that the organization of dependencies can reveal design rules which capture architects' intent. Similarly, Cai et al. [8, 37, 50, 51] proposed maintainability measurements and problematic dependency groups to detect change-prone and bug-prone files [8, 36, 37]. Lots of studies [11, 12, 14, 32] used the dependency graph to recover the design architecture. Recent studies measured the impact of dependency on software analyses [23, 25]. Inspired by them, our work proposes the DepFCD to reveal the dependency facade by representing the downstream as a dependency graph.
.

TABLE VIII: The case summary of conflict blocks on the dependency facade

| Conflict Location | Count | The Cases Regarding Downstream Modifications | Features |
|---|---|---|---|
| Method blocks | 132 | Downstream development modifies *native* method blocks: a) modify/add condition branch (45); b) modify/add call site(43); c) modify/add variable(24); d) update log(10); e) others(10). | *Call*; ⑩⑪. |
| Class Field | 31 | Downstream development modifies or adds class fields: a) class static constants with non-SDK restriction(8); b) class variable members to aggregate upstream or downstream class object(25). | *Aggregate*; ⑧; access non-SDK APIs |
| Import | 31 | Downstream development imports: a) *actively native* entity(8); b) *intrusively native* entity(3); c) static constant(5); d) others(15). | *Aggregate*, *Call*, *Override*, *Implement*; ④⑤⑥. |
| Parameters | 18 | Downstream development changes the parameters of *intrusively native* methods and calls them: a) add a new parameter(13); b) modify parameter type(5). | *Call*; ⑨. |
| Accessibility | 10 | Downstream development changes the accessibility (*private*, *protected*, and *public*) of a *native* entity. These entities can be: a) classes(2); b) methods(2); c) class fields(5). | *Call*, *Aggregate*, *Inherit*; ①. |
| Extensive method | 8 | Downstream development adds *extensive* methods for *native* classes. The methods can be: a) overriding a method due to interface implementation(3); b) adding a method in *native* interface or parent class(1); c) extracting *native* code(1); d) others(3). | *Call*, *Override*, *Implement*, *Inherit*; ⑤⑥⑮. |
| Final | 2 | Downstream development removes the *final* keyword of an entity: a) remove the *final* of a *native* class field to re-assign its value(2). | ②⑧. |
| Annotation | 2 | Downstream development removes the annotation: a) remove "@Deprecated" annotation(1), b) remove non-SDK restriction level annotation(1). | *Annotate*; ③; *unsupported* $\rightarrow$ *sdk*. |
| Inner-classes | 1 | Downstream development defines an *extensive* inner-class inside a *intrusively native* class(1). | *Call*, *Aggregate*; ⑦. |
| Upstream Update | 49 | These conflict blocks only contain modifications made by upstream development. | / |
| Others | 16 | These conflict blocks were reported near the downstream modifications. | |

▨ highlights interface-level dependencies, ▨ highlights intrusive operations encoded in Table IV, and ▨ highlights dependency constraints.

**Cross-project Dependency Management.** Cross-project dependencies have become common in modern software development, due to the reuse of existing libraries and packages [10]. Abate et al. [3] reviewed various package managers. The work of [19] examined dependency management issues for JavaScript libraries. Wang et al. [13] focused on the PyPi ecosystem. A recent work [10] detected dependency smells that have a negative impact on project maintenance. Differently, our work reveals and studies interface-level dependencies, intrusion-level dependencies, dependency constraints, and merge conflicts related to them.

**Android Changes and Merge Conflicts.** Prior works have researched changes of Android or downstream versions, such as bugs [5, 17], commit activities [44], API evolution [34], and security issues caused by non-SDK exploitation [18, 52]. The work of [31] investigated the commits of the LineageOS with upstream AOSP. Liu et al. [29] examined merge conflicts and revealed their impacts on Android mobile applications. An industrial case study by Sung et al. [46] studied the problem of upstream merge-induced conflicts for Microsoft Edge development. Our work is most similar to the work of [29, 31]. However, we aim at demystifying the dependency facade to hint at a better coupling for merge conflict reduction.
.

## VIII. Conclusion

Our work focuses on the dependency facade that couples downstream projects with the upstream Android. We propose the DepFCD to model *dependency facade* from three aspects, including *interface-level dependencies*, *intrusion-level dependencies*, and *dependency constraints*. We then empirically study the dependency facade and merge conflicts on it. Our study suggests that downstream projects should monitor the changes in the dependency facade and the violations of dependency constraints, thus helping mitigate potential merge conflicts. The study benefits both the downstream practice and other upstream and downstream ecosystems.

## Acknowledgment

This work was supported by the National Key R&D Program of China (2022YFB2703500), National Natural Science Foundation of China (62272377, 61902306, 62002280, 61721002, 61833015), and sponsored by the Honor Corporation, CCF-Huawei Open Research Fund, CCF-Mayi Open Research Fund, Fundamental Research Funds for the Central Universities (xzy012020109, xxj022019001, xzy012020009), CCF-AFSG Research Fund, China Postdoctoral Science Foundation (2020M683507, 2019TQ0251, 2020M673439), and the Young Talent Fund of Association for Science and Technology in Shaanxi, China.

## References

[1] https://en.wikipedia.org/wiki/List_of_custom_Android_distributions.

[2] Restrictions on non-sdk interfaces. https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces.

[3] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[4] AOSPA. https://github.com/AOSPA/android_frameworks_base/.

[5] Muhammad Asaduzzaman, Michael C Bullock, Chanchal K Roy, and Kevin A Schneider. Bug introducing changes: A case study with android. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 116–119. IEEE, 2012.

[6] Git Blame. https://git-scm.com/docs/git-blame.

[7] Panuchart Bunyakiati and Chadarat Phipathananunth. Cherry-picking of code commits in long-running, multi-release software. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 994–998, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Yuanfang Cai, Lu Xiao, Rick Kazman, Ran Mo, and Qiong Feng. Design rule spaces: a new model for representing and analyzing software architecture. *IEEE Transactions on Software Engineering*, 2018.

[9] CalyxOS. https://gitlab.com/CalyxOS/platform_frameworks_base/.

[10] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. Towards better dependency management: A first look at dependency smells in python projects. *IEEE Transactions on Software Engineering*, 2022.

[11] Mainak Chatterjee, Sajal K Das, and Damla Turgut. Wca: A weighted clustering algorithm for mobile ad hoc networks. *Cluster computing*, 5(2):193–204, 2002.

[12] T. Lutellier et al. Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Transactions on Software Engineering*, 44(2):159–181, 2018.

[13] Y. Wang et al. Watchman: Monitoring dependency conflicts for python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 125–135, 2020.

[14] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. Enhancing architectural recovery using concerns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 552–555. IEEE, 2011.

[15] Git. https://git-scm.com/doc.

[16] Google. https://source.android.google.cn/.

[17] 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.

[18] 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.

[19] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. Dependency smells in javascript projects. *IEEE Transactions on Software Engineering*, 2021.

[20] Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. Understanding merge conflicts and resolutions in git rebases. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 70–80, 2020.

[21] Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. Understanding merge conflicts and resolutions in git rebases. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 70–80, 2020.

[22] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. Extracting concise bug-fixing patches from human-written patches in version control systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 686–698. IEEE, 2021.

[23] Wuxia Jin, Yuanfang Cai, Rick Kazman, Gang Zhang, Qinghua Zheng, and Ting Liu. Exploring the architectural impact of possible dependencies in python software. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 758–770, 2020.

[24] Wuxia Jin, Yuanfang Cai, Rick Kazman, Qinghua Zheng, Di Cui, and Ting Liu. Enre: a tool framework for extensible entity relation extraction. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, pages 67–70. IEEE Press, 2019.

[25] Wuxia Jin, Dinghong Zhong, Yuanfang Cai, Rick Kazman, and Ting Liu. Evaluating the impact of possible dependencies on architecture-level maintainability. *IEEE Transactions on Software Engineering*, pages 1–1, 2022.

[26] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun Chen, and Jinqiu Yang. Studying test annotation maintenance in the wild. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 62–73. IEEE, 2021.

[27] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518. IEEE, 2017.

[28] LineageOS. https://github.com/LineageOS/android_frameworks_base/.

[29] Pei Liu, Mattia Fazzini, John Grundy, and Li Li. Do customized android frameworks keep pace with android? pages 1–12, 2022.

[30] MagicUI. https://www.hihonor.com/global/club/.

[31] 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.

[32] Spiros Mancoridis, Brian S Mitchell, Yihfarn Chen, and Emden R Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. Proceedings. IEEE International Conference on*, pages 50–59. IEEE, 1999.

[33] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

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

[35] MiUI. https://global.miui.com/en.

[36] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[37] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Decoupling level: a new metric for architectural maintenance complexity. In *Proceedings of the 38th International Conference on Software Engineering*, pages 499–510. IEEE, 2016.

[38] OmniROM. https://github.com/omnirom/android_frameworks_base/.

[39] ORACLE. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html#wp16437.

[40] RefactoringMiner. https://github.com/tsantalis/RefactoringMiner.

[41] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 167–176, 2005.

[42] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. Smartcommit: a graph-based interactive assistant for activity-oriented commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 379–390, 2021.

[43] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. Detecting semantic conflicts via automated behavior change detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 174–184, 2020.

[44] Vibha Singhal Sinha, Senthil Mani, and Monika Gupta. Mince: Mining change history of android project. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 132–135. IEEE, 2012.

[45] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. Mock objects for testing java systems. *Empirical Software Engineering*, 24(3):1461–1498, 2019.

[46] 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.

[47] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022.

[48] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494, 2018.

[49] Dimitrios Tsoukalas, Nikolaos Mittas, Alexandros Chatzigeorgiou, Dionisis D Kehagias, Apostolos Ampatzoglou, Theodoros Amanatidis, and Lefteris Angelis. Machine learning for technical debt identification. *IEEE Transactions on Software Engineering*, 2021.

[50] Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 967–977. ACM, 2014.

[51] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, pages 488–498. ACM, 2016.

[52] Shishuai Yang, Rui Li, Jiongyi Chen, Wenrui Diao, and Shanqing Guo. Demystifying android non-sdk apis: Measurement and understanding. 2022.