Empirical Software Engineering
https://doi.org/10.1007/s10664-020-09877-w

API compatibility issues in Android: Causes and effectiveness of data-driven detection techniques

Simone Scalabrino[1] · Gabriele Bavota[2] · Mario Linares-Vásquez[3] · Valentina Piantadosi[1] · Michele Lanza[2] · Rocco Oliveto[1]

Published online: 07 October 2020

Abstract

Android fragmentation is a well-known issue referring to the adoption of different versions in the multitude of devices supporting such an operating system. Each Android version features a set of APIs provided to developers. These APIs are subject to changes and may cause compatibility issues. To support app developers, approaches have been proposed to automatically identify API compatibility issues. CID, the state-of-the-art approach, is a data-driven solution learning how to detect those issues by analyzing the change history of Android APIs ("*API side*" learning). In this paper (extension of our MSR 2019 paper), we present an alternative data-driven approach, named ACRYL. ACRYL learns from changes implemented in apps in response to API changes ("*client side*" learning). When comparing these two solutions on 668 apps, for a total of 11,863 snapshots, we found that there is no clear winner, since the two techniques are highly complementary, and none of them provides a comprehensive support in detecting API compatibility issues: ACRYL achieves a precision of 7.0% (28.0%, when considering only the severe warnings), while CID achieves a precision of 18.4%. This calls for more research in this field, and led us to run a second empirical study in which we manually analyze 500 pull-requests likely related to the fixing of compatibility issues, documenting the *root cause* behind the fixed issue. The most common causes are related to changes in the Android APIs (∼ 87%), while about 13% of the issues are related to external causes, such as build and distribution, dependencies, and the app itself. The provided empirical knowledge can inform the building of better tools for the detection of API compatibility issues.

Communicated by: Yasutaka Kamei and Andy Zaidman

Scalabrino and Oliveto gratefully acknowledge the financial support of the Italian Ministry of Education and Research for the PON-ARS01 00860 project on "Ambient-intelligent Tele-monitoring and Telemetry for Incepting and Catering over hUman Sustainability - ATTICUS".

Bavota gratefully acknowledges the financial support of the Swiss National Science Foundation for the CCQR project (SNF Project No. 175513).

Lanza gratefully acknowledges the financial support of the Swiss National Science Foundation for the SNF-NRP 75 project on "Exploratory Visual Analytics for Interaction Graphs".

This article belongs to the Topical Collection: *Mining Software Repositories (MSR)*

✉ Simone Scalabrino
simone.scalabrino@unimol.it

Extended author information available on the last page of the article.

Keywords Android · API compatibility issues · Empirical study · Taxonomy

1 Introduction

The fragmentation of the Android ecosystem is well documented in the literature (Han et al. 2012; Zhou et al. 2014; Wei et al. 2016; Joorabchi et al. 2013; Choudhary et al. 2015; Linares-Vásquez et al. 2017). The high number of hardware devices supporting Android, the fast evolution of the Android APIs/OS, and the existence of customized versions of the APIs/OS deployed by Original Equipment Manufacturers (OEMs) on their devices, leads to a vast number of possible running environments for an app (Han et al. 2012; Zhou et al. 2014; Mutchler et al. 2016). Changes in the Android APIs are the rule rather than the exception (Linares-Vásquez et al. 2013; McDonnell et al. 2013 ; Linares-Vásquez et al. 2014; Bavota et al. 2015), pressuring developers to quickly react to newly released APIs to avoid issues related to *API compatibility issues*.

To illustrate a concrete example of API compatibility issue, let us introduce the GitHub issue 5059 of the `libgdx` framework[1] (fixed in commit 88e0b2e). The issue states that some sounds are not played anymore, because of changes to the Android API. Therefore, conditional statements should be added to let the app know the Android version executed by the device, to adapt its behavior. This type of compatibility issues is common and inspired approaches to automatically detect such issues (Wei et al. 2016; Wu et al. 2017; Luo et al. 2018; Li et al. 2018a).

Approaches that detect API compatibility issues by relying on detection rules are either hand-crafted or automatically mined from the API documentation. This latter is the solution adopted by CID (Li et al. 2018a), the state-of-the-art tool using a data-driven solution able to learn how to detect API compatibility issues by analyzing the changes in the history of Android APIs ("API side" learning). We define an approach *data-driven* if it is able to automatically infer rules from existing data (i.e., not hand-crafted by human experts). While CID is able to detect these issues, it lacks patch suggestion.

We propose a data-driven solution, ACRYL, which adopts a different approach: it learns from changes implemented in other apps in response to API breaking changes ("client side" learning). This allows ACRYL to (i) recommend how to fix the detected issue, and (ii) identify suboptimal API usages in addition to API compatibility issues. With "suboptimal API usages" we refer to cases in which an app is using an API available in all the versions supported by the app (thus not being a compatibility issue) but that, starting from a specific version, can be replaced by a newly introduced API better suited for the implemented feature.

We run an empirical study involving 11,863 snapshots of open source Android apps to compare the two data-driven solutions, and we found that none of them is superior to the other. Indeed, we found CID and ACRYL to be highly complementary, i.e., they identify an almost disjointed set of compatibility issues. Both techniques have substantial limitations, indicating the need for more research in this field aimed at fostering the development of better tools for API compatibility detection.

As a consequence of our findings, we run a qualitative study investigating the root causes behind the API compatibility issues fixed by developers. This study has the goal of providing empirical evidence useful for the research community to identify limitations of existing

[1] https://github.com/libgdx/libgdx/issues/5059

detection tools and, as a consequence, develop better solutions. The study has been run by manually analyzing 500 pull requests automatically selected as likely related to the fixing of API compatibility issues in Android apps. As output of this study, we present a detailed taxonomy of root causes behind compatibility issues (Fig. 9), discussing interesting examples, the solution adopted by developers for their fixing, and directions/challenges for future research work in the area. Besides, we estimate the *recall* of the tools for each main category of the taxonomy to identify problems that state-of-the-art approaches fail to identify. We found that the two experimented approaches struggle at finding most of the issues that developers fixed and issues belonging to some categories (i.e., "Support for new Android feature", "GUI handling", and "Energy saving") are never detected.

This paper is an extension of our previous work (Scalabrino et al. 2019) published at the 16th International Conference on Mining Software Repositories, MSR 2019, in which we presented and evaluated ACRyL. The qualitative study represents its main novel contribution.

We release all the data used in both studies as part of a replication package (Scalabrino et al. 2020), which also includes our tool, ACRyL.

Paper Structure Section 2 presents background information on API compatibility issues and discusses the related works. Section 3 presents ACRyL, while Section 4 describes the design and the results of the empirical study aimed at comparing ACRyL with CiD. Section 5 presents the qualitative study investigating the root causes behind API compatibility issues fixed by Android developers. Section 7 concludes the paper after a discussion of the threats to validity (Section 6).

2 Background & Related Work

We describe how developers deal with compatibility issues in Android apps. Afterwards, we overview the related literature.

2.1 Handling API Compatibility Issues in Android

Android apps can be used in devices running different versions (levels) of the Android API. To deal with compatibility issues, app developers can define a range of API levels they officially support. This is done by setting two attributes in the Android manifest file: `android:minSdkVersion` and `android:targetSdkVersion`.[2] By defining those attributes, developers indicate to the Google Play store which devices can install an app.[3]

Each version of the Android API can include changes impacting, more or less severely, the apps' code. This includes changes to API usage patterns, deprecated APIs, new APIs (possibly replacing the deprecated ones), and removed APIs, generally already deprecated a few versions earlier. Therefore, in addition to the SDK-version attributes in the manifest, Android developers generally include in their apps code implementing Conditional API Usages (CAUs), as in the example reported in Fig. 1.

[2] A third attribute, `android:maxSdkVersion`, does also exist, but the Android documentation recommends to not declare it, since by default it is set to the latest available API version.

[3] https://developer.android.com/guide/topics/manifest/uses-sdk-element

```
1  public void setBackground(View view, Drawable image) {
2      if (Build.VERSION.SDK_INT < VERSION_CODES.JELLY_BEAN) {
3          view.setBackgroundDrawable(image);
4      } else {
5          view.setBackground(image);
6      }
7  }
```

Fig. 1 Example of Conditional API Usage (CAU) from an Android app

CAUs are code blocks that check the current Android version on which the app is running and, based on the result of this check, establish the code statements to execute, including invocations to specific APIs. For example, if the Android version is lower than X, $\texttt{API}_i$ is invoked, otherwise, a call to $\texttt{API}_j$ is performed. The version of the API on which the app is running is identified at runtime by using the `VERSION_SDK_INT` global attribute or the specific constant available for each level of the Android API (e.g., `VERSION_CODES.JELLY_BEAN`).[4] CAUs can be used to handle different types of compatibility issues related to backward (i.e., potential problems with older SDK versions) and forward (i.e., potential problems with new SDK versions) compatibility.

Some compatibility issues cannot be handled using CAUs. Consider, for example, the class `Fragment`, which represents a reusable part of an Activity. Such a class was introduced in API level 11 (Android 3.0): if developers wanted to use such a class, they would need to drop the compatibility for the incompatible versions. Indeed, to create a `Fragment`, developers need to extend such a class, and this cannot be handled with a CAU. For this reason, the Android APIs include a support library[5] to provide a compatibility layer that allows developers to normally use framework-related provided in newer releases without limiting the support to older versions.

2.2 Related Work

Previous research has been done on API misuses — e.g., Amann et al. (2016, 2018) — and deprecated APIs (Robbes et al. 2012; Brito et al. 2016; Sawant et al. 2016; Zhou and Walker 2016; Li et al. 2018b). We focus our discussion here on works related to Android APIs. The problem of API-induced issues in Android apps has been widely discussed by practitioners and researchers. For example, the change- and fault-proneness of Android APIs have been shown to have a direct impact on the apps quality as perceived by users (McDonnell et al. 2013 ; Linares-Vásquez et al. 2014; Bavota et al. 2015).

Besides the change- and fault-proneness of APIs, the problem of inaccessible Android APIs has also been recently studied by Li et al. (2016). An API is defined as inaccessible when (i) it is not part of the public API, (ii) it is not hidden to developers, since it can be used via reflection-based invocations at runtime or by building customized libraries, and (iii) provides developers with features not available through any public API method. Their study shows that inaccessible APIs (i) are widely used by apps' developers, (ii) are less stable than public APIs, and (iii) do not provide guarantees in terms of forward compatibility.

Wu et al. (2017) analyzed compatibility issues in Android by conducting an empirical study to measure the consistency between the SDK versions declared by developers in the Android manifest files (i.e., the file declaring the minimum and target SDKs supported by the apps), and the APIs used in the apps. The results from the analysis of 24k apps show that (i) declaring the targeted SDK is not a common practice, (ii) about 7.5% of the apps

[4] https://developer.android.com/reference/android/os/Build.VERSION_CODES

[5] https://developer.android.com/topic/libraries/support-library

under-set the minimum SDK versions (i.e., they declare lower versions than the minimum required by the used APIs), and (iii) some apps under-claim the targeted SDK versions (i.e., the developers pick targeted versions above the one supported by the used APIs).

Luo et al. (2018) focused on API misuses in terms of outdated/abnormal APIs (i.e., whether apps use APIs with the `@deprecated`, `@hide`, and `@removed` annotations). Their study show that 9k+ out of 10k analyzed apps suffer from misuses with outdated/abnormal APIs.

Dilhara et al. (2018) presented ARPDroid, an open-source tool to find and repair incompatible permissions used in a given app. In a related research thread, Zhang and Cai (2019) searched into developers' intentions to achieve apps compatibility and to avoid potential compatibility issues. By comparing benign and malicious apps, they found that 0.32-1.24% benign apps and 0.39-16.88% malware apps do not specify `android:minSdkVersion` (which Android recommends to specify), while 0.52-6.1% benign apps and 0.14-2.71% malware apps define instead the `android:maxSdkVersion` which, as previously explained, Android does not recommend to specify.

Related to compatibility issues is also the work by Fazzini and Orso (2017) presenting DiffDroid. The proposed technique supports developers in the identification of cross-platform inconsistencies (i.e., CPIs) in mobile apps. DiffDroid combines input generation with differential testing to detect inconsistencies in the behavior exhibited by an app on different platforms.

We are more interested in compatibility issues due to public deprecated/removed APIs, or to device specific compatibility issues introduced by OEMs when modifying the original Android APIs and OS. The seminal work by Wei et al. (2016) represents a first effort to provide developers with a solution for detecting compatibility issues. The authors manually analyzed the source code of 27 apps looking for code patterns used by developers to fix/deal with compatibility issues. Then, the patterns where codified into rules that were implemented in a tool called FicFinder. While FicFinder had the merit to start the work on the automatic detection of API compatibility issues, it relies on 25 manually decoded rules, that can easily become obsolete.

Li et al. (2018a) propose an automatic approach based on static analysis on the app and Android APIs code to detect potential backward/forward compatibility issues. Their approach, named CiD, mines the history of Android OS to identify the lifetime of each API (i.e., the set of versions in which each API is available). Then, CiD extracts from an app under analysis a code conditional call graph that (i) links app methods to API calls, and (ii) records API level conditional checks in the graph edges. The goal is to identify API invocations in the app that might result in compatibility issues (e.g., an app declares to support the Android APIs from version 11 to 23, and uses without conditional checks an API that has been deleted in version 15). CiD is the first example of data-driven approach to detect API incompatibilities in Android and, as shown in the extensive evaluation reported by Li et al. (2018a), it ensures superior performance as compared to FicFinder (Wei et al. 2016). He et al. (2018) introduced IctApiFinder, a tool which, similarly to CiD, learns from the evolution of Android APIs and detects potential issues relying on inter-procedural data-flow analysis to reduce the number of false-positives. Both such approaches learn rules from the *API-side*.

As for security-related issues, Bartel et al. (2012) detected permission gaps using static analysis. Analyzing two datasets of Android applications, the authors show that some applications suffer from permission gaps. Besides, Backes et al. (2016) used a static runtime model to study the internals of the application framework to provide a classification of its protected resources.

2.3 The Android Studio Lint Tool

Android Studio provides a Lint tool specifically designed to detect generic issues in Android apps. Such a tool uses static code analysis to detect common anti-patterns and bad practices, and can detect nine categories of warnings: *accessibility*, *compliance*, *correctness*, *internationalization*, *interoperability*, *performance*, *security*, *testing*, and *usability*. For each category several rules are checked, including rules designed to detect possible compatibility issues in apps. We describe each of them below:

TargetSdkVersion Soon Expiring/No Longer Supported Google Play requires (as of August 2018) that apps target at least API level 26. These two rules check if the app targets previous versions.

Method conflicts with new inherited method Android SDK developers may introduce methods to already existing classes in specific versions. For example, the method m is introduced in the class C in API level 24. It is possible that the developers of a given app that targeted API levels lower than 24 defined a class for their app, C_s, which extends C and independently defined m in such a class. If the app starts targeting API level 24 or above, the app has an unintentional override of m in C_s. This rule checks such cases.

Minimum SDK and target SDK attributes not defined This rule checks whether the app defines both the `targetSdkVersion` and `minSdkVersion` in the manifest.

API Version Too Low This rule checks whether the `minSdkVersion` can be incremented without noticeably reducing the number of devices supported by the app.

Target SDK attribute is not targeting latest version This rule simply checks if the `targetSdkVersion` is the latest one to avoid that compatibility modes are used to run it and, thus, to improve its performance in general.

Calling new methods on older versions This rule checks if a method used in the app does not exist in a supported API level. Indeed, if this happens and the app is run on an older version which does not provide such a method, the app crashes.

Not overriding abstract methods on older platforms In time, new SDK versions started to provide default implementations for some methods that used to be abstract in older SDK versions. If an app targets older versions that still have abstract methods, it should provide its implementation, otherwise the app will crash. This rule checks that all the abstract methods of all the targeted versions are implemented by the app.

Using inlined constants on older versions If an app references constants that were introduced in later versions, it might happen that the app has unexpected behaviors on versions that do not define such constants. By default, the constant values are inlined (i.e., the constant references will be replaced by their actual values at compile time), but some problems may still manifest in specific contexts.

Obsolete SDK_INT Version Check It may happen that CAUs previously introduced in the code to support some specific API levels are no longer needed because the `minSdkVersion` was increased. For example, if some APIs are called when the API level is higher than 20 and the `minSdkVersion` is 21, the check is useless, since the condition will be always true. This rule check such cases.

As it can be noticed, each rule checked by the Lint tool provided by Android Studio is very specific. The only rule that may check problems similar to the ones checked by state-of-the-art tools, such as CID, and the tool we define in this paper, ACRYL, is "*calling new methods on older versions*". Such a rule, however, mostly rely on the developers' indication to work properly: if a method x contains an incompatible call and the method y, the

only one which calls x, correctly calls x inside a CAU, the Lint tool still raises a warning. To avoid this, the documentation suggests to manually annotate methods such as x with `@TargetAPI`. However, it may happen that, in a future version, another method z starts calling x outside a CAU: in such a scenario, the Lint tool would not raise a warning because of the annotation. The data-driven solutions described in the literature and the tool introduced in this paper are, by design, complementary to Lint: while Lint detects important warnings that data-driven solutions do not aim at finding (such as "*method conflicts with inherited method*"), it is designed to mostly detect simple cases, while data-driven solutions provide support for more complex scenarios, like the one previously described.

2.4 The Present Work

We compare an *API-side* approach, CID, with the data-driven approach (ACRYL) we devised to overcome some of CID's limitations.[6] While addressing the same problem using a data-driven solution, ACRYL adopts a different approach allowing it to identify suboptimal API usages in addition to API compatibility issues, and to also recommend to developers how to fix the detected issue relying on the codebase of other apps (*client-side* approach). With "suboptimal API usages" we refer to cases in which an app is using an API available in all the versions supported by the app (thus not being a compatibility issue) but that, starting from a specific version, can be replaced by a newly introduced API better suited for the implemented feature. To give a concrete example, the APIs `Bitmap.getRowBytes()` and `Bitmap.getHeight()` can be used to compute the total number of bytes in a bitmap. In API level 12, the method `Bitmap.getByteCount()` has been introduced specifically for this computation, providing a more convenient and clean way of counting the bitmap's byte.

CID cannot detect these suboptimal API usages and recommend proper refactoring actions, while ACRYL provides full support for them. In addition to that, ACRYL is able to identify compatibility issues potentially involving multiple APIs (i.e., API patterns such as the invocation of `Bitmap.getRowBytes()` and `Bitmap.getHeight()`), while CID only warns developers when a single API call represents a potential compatibility issue. These ACRYL's advantages over CID are brought by the fact that ACRYL learns from CAUs already defined by developers in a large set of apps. Thus, it can not only learn the problem (i.e., the API incompatibility being addressed with the CAU) but also the solution (i.e., how to handle it in the code). As we show in our empirical comparison, these advantages do not come for free, since ACRYL misses many relevant API incompatibility issues identified by CID. Our study shows that the two approaches are highly complementary.

An additional contribution of our work is a qualitative study investigating the root causes behind API compatibility issues fixed by Android developers. To the best of our knowledge, this is the first study providing qualitative evidence of the types of API compatibility issues experienced in Android apps.

3 Approach

We propose ACRYL (Android Client-side Rule Learner), an approach and a tool to automatically detect API compatibility issues and suboptimal usages in Android apps. ACRYL

[6] We used CID instead of ICTAPIFINDER since it is publicly available.

```
1 public boolean isMarshmellow() {
2     return (Build.VERSION.SDK_INT >= 23);
3 }
```

Fig. 2 Example of method to check the SDK version

is a data-driven approach that relies on CAUs already defined by developers in a large set of apps (*client-side*). ACRyL works in three steps. First, it extracts information about CAUs from a given reference set of Android apps. Once the set of CAUs is extracted, ACRyL uses them to infer detection rules and assigns a confidence level to each of them based on the number of apps from which the rule is learned. Finally, the rules can be used to detect suspicious API usages in a given app.

3.1 Step 1: Extraction of Conditional API Usages (CAUs)

To extract CAUs, it is necessary to detect the conditional statements that check the current platform version (e.g., `if(version < X)`) and, then, the APIs used in its branches (e.g., if the condition is true, use $\texttt{API}_i$, otherwise use $\texttt{API}_j$). Both these tasks are not trivial and pose many challenges.

As explained in Section 2, the Android APIs provide the `Build.VERSION.SDK_INT` field to check the SDK version of the software currently running on the hardware device. Thus, looking for conditional statements checking the value of this field might seem sufficient to identify the CAUs entry points. However, developers may create utility/delegate functions to get the value of the `SDK_INT` field or to check whether the app is running on a specific SDK version.

Figure 2 shows an example of utility method we found in the analyzed apps to check whether the SDK version is greater or equal than 23 (i.e., the Marshmellow Android version). The usage of methods like `isMarshmellow()` in a conditional statement allows for checking the `SDK_INT` value without explicitly referring to it.

Assuming the ability to correctly identify the conditional statements checking (directly or indirectly) the `SDK_INT` value, it is not sufficient to look into the body of the `if/else` branches to detect the API usages, since they may contain arbitrarily deep calls to methods that only at some point use Android APIs: if the `else` branch contains an invocation to method $\texttt{M}_i$ that invokes method $\texttt{M}_j$, and this latter invokes the Android API $\texttt{A}_i$, we must be able to link the usage of $\texttt{A}_i$ to the non-satisfaction of the `if` conditional statement.

We use the following approach to detect CAUs. Given the APK of an app, we convert it to a jar using DEX2JAR[7]. Then, we use the WALA[8] library to analyze the obtained Java bytecode. In particular, each method of the app is analyzed to flag the ones (i) containing a conditional statement checking the value of the `SDK_INT` field, and (ii) having a return value depending on the result of such a checking. For example, the `isMarshmellow()` method in Figure 2 would be flagged in this phase, since it returns `true` if `SDK_INT >= 23` and `false` otherwise. This step aims at identifying all sorts of "utility methods" that can be defined by the developers to check the `SDK_INT` value. In this step we also flag methods in which `SDK_INT` is assigned to a variable and, then, the variable is used in the conditional statement. For each flagged method, we store the mapping between the returned value and the value of the condition. In our example, given a method

[7] http://code.google.com/p/dex2jar
[8] http://wala.sourceforge.net/

invoking `isMarshmellow` and using its return value in a conditional statement, we know that the condition will be true if the app is running on `SDK_INT >= 23`. In addition to literal int values, the `VERSION_CODE` Android constants are also used by developers in compatibility checks.

With this information at hand, in the second step of our analysis we re-analyze all methods in an app with the goal of extracting the CAUs. Here we define a CAU as a triplet (C, A_t, A_f), where C is the compatibility condition, and A_t and A_f are the sets of Android APIs called if C is `true` or `false`, respectively. For a given triplet, A_t or A_f can be an empty set (e.g., in case an API is invoked if a condition is satisfied, while no invocations are done otherwise). For each method in the app, we check whether it invokes one of the previously flagged utility methods in a conditional statement or in an assignment expression that is then used in the condition, e.g., `boolean isCompatible = isMarshmellow(); if(isCompatible){...}`. If this is the case, the condition in the method is "normalized" to a standard form using the corresponding `SDK_INT` value in the conditional statement: `if(SDK_INT⟨relational_operator⟩⟨int_literal⟩)`. For example, the previous conditional statement accessing the `isCompatible` variable is converted to `if(SDK_INT >= 23){...}`. Once the method is normalized, we perform interprocedural analysis of the conditional statement branches (e.g., `if`/`else` branches) identifying all the calls to Android APIs[9] and to collect the signatures of the calls (return type, API class, API method, and arguments).

We then convert all triplets in the form `SDK_INT <= X`. This means that for a triplet having its condition C as $> X$ we invert the condition ($\leq X$) and we swap A_t and A_f.

At the end of the process, we obtain a set of triplets (C, A_t, A_f), with $|A_t| \geq 0$ and $|A_f| \geq 0$. Note that, because of the interprocedural analysis, it is possible that many API calls are included in A_t and/or A_f, even if only a few of them require the CAU. In this step we keep the whole sequences, that will be later refined (see Section 3.3).

We do not consider CAUs with a condition check in the form `if(version` $!=$ `X)`, because these rules are generally app specific and their meaning depends on the $MinSDK$ version declared by the apps. To clarify, a CAU (`if(version != 11)`, $\texttt{API}_\texttt{i}$, $\texttt{API}_\texttt{j}$) can have two different meanings in an app declaring $MinSDK = 11$ and in an app declaring $MinSDK = 4$. In the first case, the CAU is probably needed because versions older than 12 need to invoke $\texttt{API}_\texttt{j}$, i.e., it is equivalent to the CAU (`if(version <= 11)`, $\texttt{API}_\texttt{j}$, $\texttt{API}_\texttt{i}$). However, since the only version older than 12 that is supported by the app is 11, the developer used the check in the form $!= 11$. In the second case ($MinSDK = 4$), the developer is instead using the check to customize the behavior of the app on a specific version (11) among the ones supported by the app. Thus, in this case, the checked condition is not equivalent to `if(version <= 11)`. We preferred to only learn CAUs that are more likely to represent general issues related to specific SDK interval versions, i.e., the ones in the form (`if(version <= X)`, $\texttt{API}_\texttt{i}$, $\texttt{API}_\texttt{j}$), and that can be more easily generalized.

3.2 Step 2: Inferring Compatibility Detection Rules

Given the set of CAUs represented as triplets and extracted from hundreds of apps, we define a *detection rule* as a CAU that appears in a set of apps S. We define S the *support* of the rule. To verify whether a CAU appears in multiple apps, we first clean and standardize all extracted CAUs.

[9]We identified Android APIs by checking the package the class implementing the API comes from. The list of packages we consider as part of the Android APIs is available in our replication package.

The pre-processing phase consists in removing noisy Android APIs that do not bring information useful for the extraction of meaningful rules. We filter out from A_t and A_f all the logging APIs, e.g., a triplet ($\leq$ 24, `Log.w`, `Activity.requestPermissions`) becomes equivalent to ($\leq$ 24, $\emptyset$, `Activity.requestPermissions`). We also exclude all calls to `android.content.Context.getString(int)` and `android.content.Context.getSystemService(String)`, since these methods are quite generic and appear in many of the CAUs we extracted, but with different "semantics". For example, `getSystemService` returns the "handle to a system-level service by name". This method supports "taskmanager" as parameter **value** since SDK level 21. Therefore, some apps may have a check before calling such a method with that specific parameter value. However, ACRyL extracts rules considering the complete signature of the method (including the parameter type), but ignoring the parameter value. While considering the parameter value is an option, this would not allow to carefully assess the number of apps in which a CAU appears, since two identical CAUs with different parameter values will be considered unrelated.

Thus, if we consider `getSystemService` in the extracted CAUs, ACRyL would create a rule raising a warning when `getSystemService` is invoked without checking for a SDK level higher or equal than 21, creating many false positives (e.g., the parameter value "alarm" is supported since the first version of the APIs). Once these APIs are removed, the pre-processing ends with the removal of all CAUs having $A_t = A_f$ (i.e., the set of APIs invoked is exactly the same independently from the result of the condition check). This is possible in two cases. First, A_t and A_f differ only for the usage of one of the three APIs we ignore (e.g., A_t includes logging statements, while A_f does not). Second, A_t and A_f differ for the value of the parameters passed at runtime that, as said, is ignored by ACRyL. Finally, we aggregate all the equivalent CAUs and we define the detection rules as pairs (CAU, S), where S is the set of apps in which the CAU appears. S is used to compute the confidence level for the rule as described in Section 3.3.

3.3 Step 3: Rules Definition and Confidence Level

The intuition behind the confidence level is that if a rule appears in many apps, it is likely to be meaningful and useful to spot real issues. We do not consider the number of times that a rule appears inside a single app as a good indication of its reliability, since the same developer could apply a wrong rule multiple times in her app.

Given a rule $R_i = ((C_j, A_t, A_f), S)$, we do not compute the confidence level by simply counting the number of apps in which its CAU (i.e., C_i, A_t, A_f) appears, since this results in a strong underestimation of the actual importance of the rule. Consider the case in which we have just two rules: $R_1 = ((\leq 20, \{A, B, C\}, \{X, Y\}), \{\alpha_1, \alpha_2\})$ and $R_2 = ((\leq 20, \{A, C\}, \{Y\}), \{\alpha_3, \alpha_4, \alpha_5, \alpha_6\})$. Here A, B, C, X, and Y represent five different Android APIs, and $\alpha_1, ..., \alpha_6$ represent six Android apps. Since the condition checked in the two rules is the same and R_2 is "contained" in R_1 (i.e., APIs in R_2's A_t are contained in R_1's A_t, and the ones in R_2's A_f are contained in R_1's A_f), every R_2 instance is also a R_1 instance. Therefore, by counting frequencies individually, R_2 does not appear in only four apps, but in six apps ($\alpha_1 ... \alpha_6$). Also, it is sufficient to look for instances of R_1 in order to detect issues of the type R_2, since R_1 is a generalization of R_2. In other words, R_2 is likely an instance of R_1 customized for a specific app. For this reason, we use the following procedure to compute the confidence level of each rule.

First, we formally define the relationship "is generalization of" as ($\prec$): $R_1 = ((C_1, A_{t1}, A_{f1}), S_1) \prec R_2 = ((C_2, A_{t2}, A_{f2}), S_2)$ if $C_1 = C_2$, $A_{t1} \subseteq A_{t2}$, $A_{f1} \subseteq A_{f2}$,

and $|S_1| \geq |S_2|$. If R_1 appears in less apps than R_2 (i.e., $|S_1| < |S_2|$), the relationship does not hold. Indeed, even if more generic, R_1 may have been introduced by mistake in some apps, and the fact that the specific rule is more popular may indicate that it is the correct way of implementing the CAU. Also, we consider rules with empty sets for A_t or A_f as special cases; the generalization relationship ($\prec$) for rules containing empty sets holds only if the empty set is present in both the rules in the same branch.

For example, the rule $R_1 = ((\leq 20, \emptyset, \{Z\}), S_1)$ is not a generalization of the rule $R_2 = ((\leq 20, \{A, B\}, \{Z, Y\}), S_2)$.

The fact that a rule does not include any alternative API can have a completely different semantic. Consider the CAUs (≤ 15, `View.setBackgroundDrawable`, `View.setBackground`) and ($\leq 15, \emptyset$, `View.setBackground`): while the first expresses the alternative usage of two APIs, the second might have been introduced because some apps decided to use an image as a background just for specific versions. Starting from this definition, we create a Directed Acyclic Graph $G = (R, E)$, where R is the set of rules (i.e., nodes), and E is the set of $\prec$ relationships existing between rules (i.e., edges). For each pair of rules $< R_1, R_2 > \in R$, we create an edge going from R_1 to R_2 if $R_1 \prec R_2$. We consider all the connected sub-graphs $\rho \in G$. For each ρ, we keep the root of the sub-graph (i.e., the most generic rule) and we compute its confidence level as $|\bigcup_{(CAU_i, S_i) \in (\rho)} S_i|$, i.e., the cardinality of the set composed by the union of the apps in which the generic rule and its "child rules" appear. These are the detection rules ACRyL uses to identify compatibility issues. The more specific rules are removed, since (i) contained in the more general root rules, and (ii) unlikely to represent general compatibility issue patterns.

Figure 3 shows an example of ρ. The black boxes report the number of apps in which each rule is contained (in this example, we assume that each rule is contained in a disjointed set of apps). In this case, we keep only the rule (SDK_INT ¡= 20, {`Resource.getDrawable()`}, {`Resource.getDrawable(Theme)`}) and we compute its confidence as the total number of apps in which it and its child rules appear

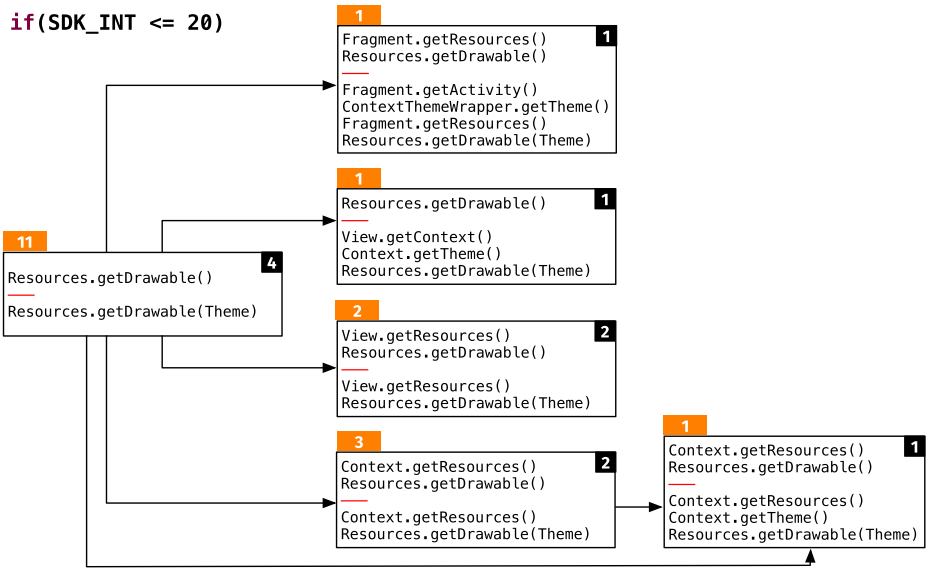


Fig. 3 Directed Acyclic Graph mapping the $\prec$ relationship

(i.e., 11) — see the orange boxes in Fig. 3. We use the confidence level as a proxy of the reliability of the rule. We conjecture that rules that appear in a sufficiently high number of apps are "reliable", i.e., they represent CAUs that should be implemented and they are not introduced by mistake. Therefore, we use a threshold, Min_{cl}, to distinguish reliable rules from unreliable ones. We only keep into account rules with confidence level higher than Min_{cl}. The tuning of the threshold Min_{cl} is presented in Section 4.1.3.

3.4 Step 4: Detecting APIs Usage Issues

We use the set of rules inferred from Step 2 and refined in Step 3 to detect potential API compatibility issues and suboptimal usages. Given an app P to check and the set of rules R, ACRyL analyzes P's bytecode and, for each $R_i \in R$, looks for usages of the R_i's A_t (or A_f) in P that are not "checked" by R_i's C in P, i.e., the APIs used in P do not have the compatibility check expected for those APIs accordingly to R_i. This, combined with the analysis of the `android:minSdkVersion` declared by P in the manifest file (see Section 2), allows ACRyL to detect the types of API issues and suboptimal usages described in Table 1. Table 1 assigns a name to each of the potential issues detected by ACRyL, and it shows the heuristic we use to detect it. When reading the table, it is important to remember that, since all the checking conditions in the detection rules have been normalized in the form `if(SDK_INT <= V`, A_t are the APIs that should be used when using V or older SDK versions (i.e., `SDK_INT <= V` is true), while the A_f should be used for newer versions. We use the lifetime model extracted by CID to determine when an API was introduced and if/when it was removed. We use such information to determine the *severity* of a warning (among *bug*, *bad smell*, and *improvement*). We briefly describe each type of potential issue we detect in the following. To ease the description, we assume that the issue has been detected with a rule having a high support and featuring the condition $C = (\leq 20, \texttt{API}_1, \texttt{API}_2)$, thus having A_t=$\texttt{API}_1$ and A_f=$\texttt{API}_2$.

Table 1 Types of API compatibility issues and suboptimal usages detected by ACRyL

Type of issue		Detection heuristic
Backward	Bug	APIs A_f are used without a compatibility check and any API in A_f does not exist in $MinSDK$ version.
	Improvement	APIs A_f are used without a compatibility check, all APIs in A_f exist in $MinSDK$, $V < MinSDK$, and any API in A_f do not exist before V.
Forward	Bug	APIs A_t are used without a compatibility check and any API in A_t does not exist in the latest SDK version.
	Bad Smell	APIs A_t are used without a compatibility check and any API in A_t is deprecated in the latest SDK version.
	Improvement	APIs A_t are used without a compatibility check but no API in A_t is deprecated or removed in the latest SDK version.
Wrong Precond. Checked		APIs A_f and A_t are used with a compatibility check, but the checked version is not the expected one V.

An issue identified with a detection rule $R = ((C, A_t, A_f), S)$ is classified into one of the supported types according to the reported heuristic; V indicates the API level subject of the condition C and $MinSDK$ the minimum SDK version supported by the app under analysis

Backward Compatibility Bug An app invokes $\texttt{API}_2$ without a compatibility check, and $\texttt{API}_2$ does not exist in the $MinSDK$ version (e.g., 18) declared in its manifest file. The C conditional check should be added to invoke $\texttt{API}_2$ only if the app is running in the versions in which $\texttt{API}_2$ is available. This is a severe bug resulting in the crash of the app.

Backward Compatibility Improvement An app invokes $\texttt{API}_2$ without a compatibility check. $\texttt{API}_2$ exists in the $MinSDK$ version declared by the app, thus does not resulting in a crash. Indeed, the rule refers to a check needed for apps running in versions older than $MinSDK$ (i.e., $MinSDK$ ¿ 20 in our running example), in which $\texttt{API}_2$ does not exist. Addressing this warning by implementing C could help the developer to improve the backward compatibility of the app (i.e., the part of the code using this API will become compatible with older, currently unsupported, versions).

Forward Compatibility Bug An app invokes $\texttt{API}_1$ without a compatibility check and $\texttt{API}_1$ does not exists in the latest SDK version. The C conditional check should be added to invoke $\texttt{API}_1$ only if the app is running in the (older) versions in which $\texttt{API}_1$ is available; $\texttt{API}_2$ should be invoked otherwise. This bug results in the crashing of the app.

Forward Compatibility Smell An app invokes $\texttt{API}_1$ without a compatibility check and $\texttt{API}_1$ exists in the latest SDK version but is deprecated. Thus, $\texttt{API}_1$ could be deleted in the future resulting in a bug. The developer can implement C invoking $\texttt{API}_2$ on the newer SDK versions, thus avoiding future bugs.

Forward Compatibility Improvement An app invokes $\texttt{API}_1$ without a compatibility check and $\texttt{API}_1$ exists in the latest SDK version and is not deprecated. However, many apps use C when accessing $\texttt{API}_1$. This might be an indication that a better API ($\texttt{API}_2$) has been introduced in newer SDK versions to accomplish the tasks previously performed using $\texttt{API}_1$. For example, one of the rules ACRyL identified is ($\leq$ 11, {`Bitmap.getRowBytes()`, `Bitmap.getHeight()`}, {`Bitmap.getByteCount()`}). The `getByteCount` API has been introduced in version 12, and returns the total number of bytes composing a `Bitmap`. This task was previously performed by using the `getRowBytes` and the `getHeight` APIs that have not been deleted or deprecated (since they are still used to accomplish specific tasks). Implementing C in this case allows to take advantage of improvements (e.g., better performance) ensured by the latest introduced APIs.

Wrong Precondition Checked An app invokes $\texttt{API}_1$ or $\texttt{API}_2$ and it implements a compatibility check using a version X different than the one expected in C (20 in our running example). For example, if ACRyL finds a CAU ($\leq$ 21, $\texttt{API}_1$, $\texttt{API}_2$) in a given app it detects a wrong precondition check, since it learned from other apps that the "right check" to do is `SDK_INT <= 20`.

4 Study 1: On the Effectiveness of Data-Driven Solutions for Identifying API Compatibility Issues

The *goal* of this study is to compare two data-driven approaches for the detection of Android API compatibility issues. We focus on CiD, as state-of-the-art approach and representative of a *API-side* learning approach, and ACRyL, as *client-side* learning approach. The *focus* is on the ability of the experimented techniques to identify issues that are actually fixed by

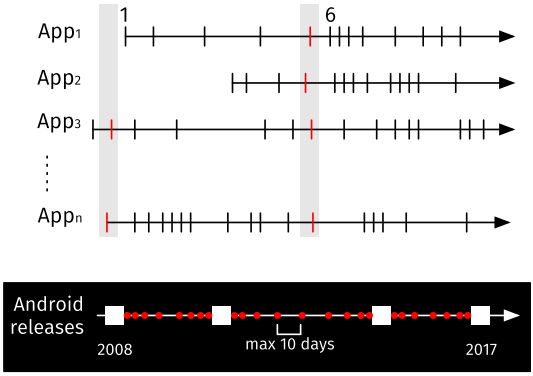


Fig. 4 Diagram of the study design. The arrows labeled with "App$_i$" represent the change history of the apps considered in our study, with the vertical lines representing the snapshots from the versioning system. We analyze apps on specific snapshots, depending on their distance from the previous/next release. For ACRyL, we only use rules inferred from past versions of the apps. We report in red, inside the grey bar, the snapshots from which the rules are learned

software developers. The *context* consists of 19,291 snapshots of 1,170 open source Android apps.

4.1 Study Design

The study addresses the following research question:

RQ$_1$**:** *What is the most effective data-driven approach to detect Android API compatibility issues?* ACRyL and CiD are compared on the basis of compatibility issues they detect in real apps and that are fixed by software developers over the apps' change history.

4.1.1 Data Collection

The first step to answer our research question is the selection of the subject mobile apps. We mined F-Droid,[10] a catalogue of free and open-source Android apps, to identify apps hosted on GitHub. This resulted in the collection of 1,170 URLs of git repositories. As explained later, we also used these apps to tune the Min_{cl} threshold aimed at excluding unreliable detection rules learned by ACRyL. We adopt the study design depicted in Fig. 4.

The arrows labeled with "App$_i$" represent the change history of the apps considered in our study, with the vertical lines representing the snapshots from the versioning system. Note that the history of the apps is not aligned, meaning that not all the apps exist in the same time period (e.g., App$_1$ was created after App$_3$ and before App$_2$). Given an app, the general idea behind our experimental design is to run ACRyL on each of its snapshots to detect compatibility issues, and then check whether the issues reported by ACRyL have been fixed by the developers in subsequent snapshots of the app. In other words, if ACRyL detects an API compatibility issue in the snapshot S_1 and this issue is fixed in S_4 by implementing a

[10] https://www.f-droid.org/

conditional API usage, we can assume that the compatibility issue detected by ACRyL was relevant. In this way, we can compute the percentage of API compatibility issues detected by ACRyL that have been fixed by the developers over the change history of the analyzed apps. This percentage will represent an underestimation of the relevance of the issues detected by ACRyL. While it is safe to assume that a fixed issue is relevant for developers, we cannot assume that a non-fixed issue is not relevant, since developers may simply be not aware of it.

Since ACRyL analyzes the code of existing apps to learn detection rules, one point to discuss is the set of apps from which the rules are learned before ACRyL can be run on a given app to analyze. Let us assume that the app under analysis is App_1 in Fig. 4. In particular, we want to run ACRyL on its first and sixth snapshot. For the first snapshot created on date d_1, we extract from each app the latest snapshot existing before d_1, and we use these snapshots to learn the rules. Then, ACRyL is run on the App_1's first snapshot with the set of rules just learned. In Fig. 4 we report in red, inside the grey bar, the snapshots from which the rules are learned. The same applies for the analysis of the sixth snapshot. In this way, ACRyL is not using "data from the future": We are simulating a real usage scenario in which the rules are learned on a set of open source apps at date d_i, and this set of rules is used to detect API compatibility issues in a date d_j ¿ d_i.

By analyzing the complete history of an app, we know the issues detected by ACRyL in each of the analyzed snapshots. Thus, we can verify whether an issue detected in snapshot S_1 has been fixed in a subsequent snapshot, allowing us to compute the fixing rate of the issues detected by ACRyL. We measure the fixing rate as $\frac{|issues_{fix}|}{|issues_{det}|}$, where $issues_{fix}$ is the number of fixed issues and $issues_{det}$ is the number of issues detected by ACRyL. A few clarifications are needed for what concerns the computation of the fixing rate. First, if the same API compatibility issue is detected in snapshots S_1, S_2, and S_3 of the same app and it is not detected anymore in snapshot S_4, we count it as **one** detected issue that has been fixed (not as three, since the issue is the same). Second, assuming again that a previously detected issue is not identified anymore in S_4, we do not consider it as fixed if the method affecting it was deleted (i.e., ACRyL does not identify the issue not because it has been fixed, but because the problematic method was deleted). In this case, we do not count the issue in the $issues_{fix}$ set nor in the $issues_{det}$ set. Indeed, we do not want to assume that the issue has been fixed/not fixed, since we do not have any evidence for that. We prefer to ignore this issue from the computation of the fixing rate to avoid introducing noise in our results. Finally, it could happen that the detection rule used in snapshots S_1, S_2, and S_3 by ACRyL to identify the issue is not part of the ACRyL's ruleset when it is run on S_4. This is a consequence of the experimental design in which, as previously explained, the set of rules used to detect issues in each snapshot may change. In this case, ACRyL will not identify the issue in S_4 not because it has been fixed, but because is not considered an issue anymore in its ruleset. For this reason, we do not consider the issue as fixed. Summarizing, a detected issue is considered fixed only if the developers added a check in the code to handle the problematic API(s) or they removed the API(s) that caused the problem in the first place.

The last thing to clarify for the adopted design is that, for a given app under analysis, we did not run ACRyL on all its snapshots. This was done because the process of re-building the ruleset for each snapshot would have been too expensive in terms of computational resources. Indeed, to build the ACRyL's ruleset to analyze a single snapshot S_1 we need, for each of the apps existing before S_1, to (i) build, using Gradle, its latest snapshot preceding S_1 and (ii) analyze its bytecode for extracting the rules. One simple option would have been to select one snapshot every n days (e.g., every 10 days). However, this would have likely

resulted in the missing of several compatibility issues that developers may have introduced and fixed within the n-day interval. Our conjecture is that compatibility issues are more likely to appear close to the release of new versions of the Android APIs. Thus, we decided to sample the snapshots to analyze by taking this into consideration. We defined a set of dates $dates$={d_1, d_2, ..., d_k} from which we extract the snapshots on which ACRyL is run for each app under analysis. This means, for example, that if an App_i is the one from which we want to detect compatibility issues, we select its snapshot closer to date d_1 (and preceding it) and we analyze it with the procedure previously explained; then, we move to the snapshot closer to d_2, and so on. This set of dates is defined in such a way that more dates are selected when approaching the dates in which new versions of the Android APIs have been released. The output of this process is depicted in the bottom part of Fig. 4, in which the white squares represent four Android API releases and the red dots are the dates selected for the analysis. The selection of the dates was performed using Algorithm 1, taking as input the dates of the Android Releases (AR). We defined as maximum interval between two subsequent dates d_i and d_{i+1} 10 days. This means that when we are far from an Android release, still we want to analyze at least one snapshot every 10 days.

Algorithm 1 Selection of the analysis dates.

```
1:  procedure ExtractDates(AR)
2:      cur ← AR_first
3:      dates ← list()
4:      while cur ≤ AR_last do
5:          append cur to dates
6:          prev ← max_i(AR_i : AR_i ≤ cur)
7:          next ← min_i(AR_i : AR_i ≥ cur)
8:          gap ← next − prev
9:          delay ← min(10, gap/10)
10:         exp ← (min(cur−prev, next−cur)) / (0.5×gap)
11:         cur ← cur + max(round(delay^exp), 1)
12:     return dates
```

We start from the date of the first stable Android release (2008/10/22) — cur in Algorithm 1, line 2 — and from an empty set $dates$ (line 3). Then, the `while` loop starting at line 4 is in charge of adding dates to the selected set until reaching the date of the last stable Android release, 2017/12/04 at the date of the experiment. In particular, the cur date is added to the set (line 5), and then the closer android release dates before and after it are stored in $prev$ and $next$, respectively (lines 6-7); gap is then used to store the days between $prev$ and $next$ (line 8) while $delay$ indicates the maximum number of days that can be skipped between $prev$ and $next$ during the analysis (line 9). It is always in the interval (0, 10], and it depends on the gap between the two releases: the larger the gap, the larger the maximum number of days that can be skipped when cur is far from the release dates. Then, we increment cur, the current date, by a value exponentially depending on distance between cur and the nearest release date (i.e., $prev$ or $next$) — lines 10-11. The closer cur to one of the release dates, the lower exp, which is always in the interval [0, 1]. In total, we considered 1,594 days, from 2008/10/22 to 2017/12/04, and we skipped 1,736 days.

By applying this process, we had to discard 502 of the 1,170 apps. This was due to (i) git repositories not existing anymore, (ii) apps not using Gradle, and (iii) apps having all builds failing using Gradle. Thus, RQ_1 is answered by considering 668 apps for a total of 19,291 built snapshots. We considered both Java and Kotlin apps, with an average size of 9.4K LOCs (measured at their latest snapshot). The first buildable snapshot for the analyzed

apps is from 2014/02/18. We release the list of apps/snapshots we considered (Scalabrino et al. 2020).

4.1.2 Data Analysis

To compare ACRyL with CID, we run this latter on the same set of apps' snapshots used to evaluate ACRyL. We run both tools on a machine with 56 cores and 396Gb of RAM. Since the code analysis performed by CID is computationally expensive, we run the tools for a maximum of 1 hour on each snapshot. If such time exceeded, we killed the process and we ignore that snapshot. We did this because, in a first attempt, we run CID without any time limit, but, on some snapshots, it run for hours, requiring a restarting of the machine. ACRyL adopts a much lighter code analysis, requiring about 5 minutes, on average, for the analysis of a single snapshot, excluding the extraction of the rules and the building time of the apps.

We answer RQ_1 by reporting the fixing rate of the issues detected by ACRyL and by CID. The comparison is done only on the set of apps on which we managed to successfully run both tools. We also discuss the fixing rate of the issues detected by ACRyL when considering all the apps on which it was run (thus not only those on which also CID worked).

Also, since the goal of our study is to compare different data-driven approaches, we analyze the complementarity of ACRyL and CID when detecting Android API compatibility issues. In particular, we compute the following overlap metrics:

$$fixed_{\text{ACRyL}\cap\text{CID}} = \frac{|fixed_{\text{ACRyL}} \cap fixed_{\text{CID}}|}{|fixed_{\text{ACRyL}} \cup fixed_{\text{CID}}|}\%$$

$$fixed_{\text{ACRyL}\setminus\text{CID}} = \frac{|fixed_{\text{ACRyL}} \setminus fixed_{\text{CID}}|}{|fixed_{\text{ACRyL}} \cup fixed_{\text{CID}}|}\%$$

$$fixed_{\text{CID}\setminus\text{ACRyL}} = \frac{|fixed_{\text{CID}} \setminus fixed_{\text{ACRyL}}|}{|fixed_{\text{ACRyL}} \cup fixed_{\text{CID}}|}\%$$

where $fixed_{\text{ACRyL}}$ and $fixed_{\text{CID}}$ represent the sets of compatibility issues fixed by developers and detected by ACRyL and CID, respectively. $fixed_{\text{ACRyL}\cap\text{CID}}$ measures the overlap between the set of fixed issues detected by both techniques, and $fixed_{\text{ACRyL}\setminus\text{CID}}$ ($fixed_{\text{CID}\setminus\text{ACRyL}}$) measures the fixed issues detected by ACRyL (CID) only and missed by CID (ACRyL). The latter metric provides an indication on how an API compatibility detection strategy contributes to enriching the set of relevant issues identified by another approach. More specifically, a low level of $fixed_{\text{ACRyL}\cap\text{CID}}$ would indicate that the complementarity of the two approaches is high (only a low number of issues would have been detected by both the approaches); on the other hand, a high level of such a metric would indicate that the two approaches find the same kind of issues.

Finally, we qualitatively discuss examples of relevant compatibility issues identified by one approach and missed by the other, to further investigate the possibility of combining the two experimented techniques.

4.1.3 Tuning of the Min_{cl} Threshold

We use the extracted data to firstly tune the Min_{cl} threshold, and then to answer RQ_1. In particular, we consider the first part of the analyzed history, from 2014/02/18 to 2016/06/04 (18 months before the latest analyzed day 2017/12/04) for the tuning of Min_{cl}. Here we analyzed the reliability of the rules at different Min_{cl} levels. A rule is considered to be

"reliable" if once it becomes part of the rule set (i.e., once it is learned from one or more apps), it does not disappear in the future (i.e., the apps from which it has been learned, continue to implement it). Indeed, if a learned rule is removed from the app from which it was learned, this might indicate that the rule was implemented "by mistake". We tune Min_{cl} to identify its minimum value that allows to discard unreliable rules.

4.1.4 Replication Package

The verifiability of our study is guaranteed through a publicly available replication package (Scalabrino et al. 2020) including the data used in the study as well as the ACRyL tool.

4.2 Study Results

Among the 11,863 snapshots considered for computing the results, we had to forcefully interrupt CiD 1,971 times, while we had to interrupt ACRyL 134 times (also in this case, due to the 1-hour maximum running time we set for each tool on each snapshot). We did not have any data about CiD for 98 of the 585 apps considered in our study, while we had no results from ACRyL for 69 apps. Specifically, CiD did not complete its analysis on 29 of the apps that ACRyL was able to analyze, while the opposite never occurred. We excluded the 98 apps that could not be analyzed with CiD from the comparison.

4.2.1 Tuning of the Confidence Level

Figure 5a shows, for each confidence level, the total number of disappeared rules in logarithmic scale.

No rules with confidence level higher than four disappeared in our dataset. There are, however, a few cases in which rules with confidence 3 or 4 disappear (11 in total). For example, the rule (($\leq$ 10, {}, {`Window.setFlags`}), S) was first mined on 2014/06/19 from a single app; then, it started to spread and it reached its peak on 2015/02/23, when it appeared in 4 apps (i.e., $|S| = 4$). However, after 2015/04/29 it started to be removed in such apps and it appeared the last time on 2016/05/24, when only one app implemented it. We found that `Window.setFlags` was introduced since the first version of the Android APIs; however, some of the flags that can be set (i.e., numeric constants used as parameters) were introduced later. Therefore, the check implemented by the apps referred to the parameter values used in those specific apps rather then to the usage of the API itself. Having

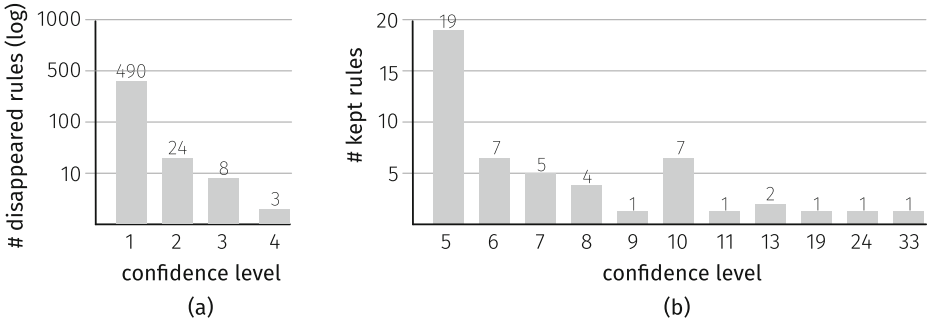


Fig. 5 Results of the tuning. On the x-axis we report different confidence levels, while on the y-axis we report the number of disappeared (a) and kept (b) rules

such a rule would have increased the number of false-positives detected by ACRyL, since it would have raised a warning in all the cases in which the API was called before version 10.

Given the achieved results, we set $Min_{cl} = 5$. We report in Fig. 5b the distribution by confidence level of the rules that did not disappear detected in the tuning time period.

4.2.2 Comparison Between ACRyL and CID

CID reported, in total, 5,926 warnings, while ACRyL reported 4,102 warnings. We noticed that there is a single app (Ultrasonic) for which ACRyL reports many warnings (2,614, 834 of which are fixed). Given the presence of this strong outlier in our dataset, we decided to exclude it from our study to avoid that a single app accounts for most of the considered data points. Excluding Ultrasonic, ACRyL raised 1,488 warnings, i.e., about a fourth of the ones reported by CID.

We report in Fig. 6 the distribution of warnings raised by ACRyL and CID. Both tools report a limited number of warnings for most of the apps. CID reports at most 10 warnings for 75% of the apps. On the other hand, because of the lower number of warnings, ACRyL reports at most 10 warnings for about 95% of the apps. ACRyL reported no warnings at all for 318 apps, while CID for 252 apps.

We report in Table 2 the comparison between ACRyL and CID, also showing the percentage of fixes for different categories of warnings. Since ACRyL reports both severe and non-severe warnings, we report the results considering (i) all the warnings, and (ii) only severe warnings. We do this because CID, by default, only reports severe warnings, which are more likely to be addressed by the developers, since they may result in actual bugs.

The overall precision of ACRyL is 7.0% (15.1% for Forward warnings and 5.9% for Backward warnings). However, if only severe warnings are taken into account, ACRyL achieves higher precision compared to CID (28% vs 18.4%). It is worth noting that ACRyL reports a very low number of severe issues (50) compare to the ones reported by CID (4,622): this means that the higher precision comes at the price of having a lower number of warnings, even possibly useful. Despite this possible limitation, we show in Section 5.2 that the estimated *recall* of the two approaches is comparable.

To analyze more in depth the distribution of the warnings raised by ACRyL, we report in Table 3 the detailed results for all the categories of warnings that ACRyL can detect.

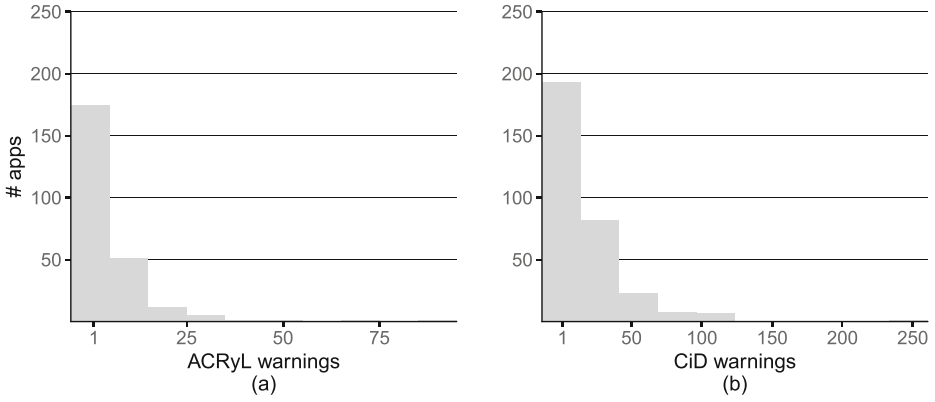


Fig. 6 Distribution of the warnings detected by the ACRyL (**a**) and CID (**b**), on the x-axis, in the analyzed apps (y-axis) — apps with at least a warning

Table 2 Comparison between ACRyL and CID

	#Fixed	#Warnings	Fix Rate
	Backward		
ACRyL (all)	71	1,202	5.9%
ACRyL (severe)	14	50	28.0%
CID	877	4,622	19.0%
	Forward		
ACRyL (all)	25	166	15.1%
ACRyL (severe)	0	0	//
CID	201	1,222	16.4%
	Total		
ACRyL (all)	96	1,368	7.0%
ACRyL (severe)	14	50	28.0%
CID	1,078	5,844	18.4%

For ACRyL, we report the results obtained considering (i) all the warnings, and (ii) only the severe ones. CID only reports severe warnings by design

We do this for (i) all the apps used for the comparison and (ii) only the apps that ACRyL could analyze. Most of the warnings found by ACRyL belong to the macro-category "Backward". Specifically, the warnings from the category "Backward Bug" are among the most frequently fixed ones (28%) as previously reported, while the backward "improvements" rarely get fixed (only 5% of times). ACRyL did not report any "Forward Bug" warning. This is probably due to the fact that Android APIs are seldom completely removed. Besides, Android provides compatibility layers that allow developers to avoid forward compatibility problems even without modifying the code, at the price of a performance overhead. We

Table 3 Performance of ACRyL by categories of warnings computed on (i) the apps analyzed by both the tools and on (ii) all the apps analyzed by ACRyL. The bottom rows report aggregated results for severe warnings (*bugs*) and non-severe warnings (*improvement* and *bad smell*)

		$Apps_{CID} \cap Apps_{ACRyL}$			$Apps_{ACRyL}$		
		#Fixed	#Warnings	Fix Rate	#Fixed	#Warnings	Fix Rate
Backward	Bug	14	50	28%	14	50	28%
	Improvement	57	1,152	5%	59	1,220	5%
	Total	71	1,202	5.9%	73	1,270	5.7%
Forward	Bug	0	0	//	0	0	//
	Bad Smell	5	5	100%	5	5	100%
	Improvement	20	161	12%	22	196	11%
	Total	25	166	15.1%	27	201	13.4%
Wrong Precond. Checked		0	0	//	0	0	//
Severe warnings		14	50	28%	14	50	28%
Non-severe warnings		82	1,318	6%	86	1,421	6%

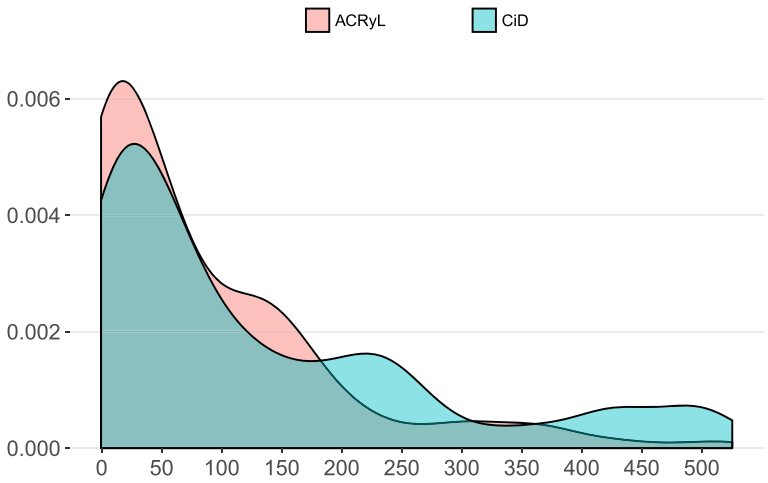


Fig. 7 Distribution of days needed to fix warnings

found, instead, that the developers fixed all the "Forward Bad Smell" warnings reported by ACRyL, which shows that this type of issues are worthwhile to detect.

We report in Fig. 7 the distribution of the fixing time (in days) for both the approaches. The fixing time indicates the "survivability" of the compatibility issue in the app. While the distribution shows a very similar trend (most of the warnings get fixed in less than 100 days), the warnings reported by ACRyL get fixed quicker (85.5 days *vs* 134.8 days, on average).

Finally, Table 4 shows the overlap metrics of the fixed compatibility issues detected by the tools. Table 4 clearly highlights that the two techniques are highly complementary. Indeed, only a small part of the warnings (0.9%) are reported by both tools, while the remaining warnings are only reported by one of the two tools. More specifically, ACRyL reports 75 new warnings that CiD did not report.

4.2.3 Examples of Warnings

An example of warning detected by both the approaches concerns the `Kandroid` (GH: andresth/Kandroid) app. ACRyL reported a "Backward Improvement" warning when analyzing the snapshot from 2017/03/02. Then, on 2017/04/25, the developers reduced the `minSDK` (from 21 to 17) to improve the compatibility of their app; however, doing so, the

Table 4 Overlap between ACRyL and CiD

	$\text{fixed}_{\text{CiD}\setminus\text{ACRyL}}$	$\text{fixed}_{\text{ACRyL}\setminus\text{CiD}}$	$\text{fixed}_{\text{CiD}\cap\text{ACRyL}}$
Backward	887 (93.0%)	56 (5.9%)	11 (1.2%)
Forward	201 (91.4%)	19 (8.6%)	0 (0.0%)
Total	1,088 (92.7%)	75 (6.4%)	11 (0.9%)

warning became a critical bug, since the API they used was not supported in Android versions before 21. This was fixed on 2017/05/01.[11] CID was able to catch the same problem, but only when it became a critical bug (i.e., when developers reduced the `minSDK` value).

An example of warning detected only by CID is from `Dandelion` (GH: gsantner/dandelion). Such a warning belongs to the category "Forward". Even if the line of code concerned was present since the first commit of the app, CID raised the warning for the first time on 2016/06/07, probably because the API was still supported until then. The bug was fixed on 2016/06/09.[12] ACRYL did not learn at all a rule for this API, since only a few apps implemented a CAU for it.

Finally, an example of warning detected only by ACRYL concerns the `Rick App` (GH: no-go/RickApp). ACRYL reported a Backward Bug on 2016/12/11. Such a warning was present since the first release of the app and it was fixed on 2016/12/17.[13] This bug involved Android versions below 4.4.4 and the author explicitly says in the README that the app was not tested for such Android versions.

4.3 Discussion

Both *API-side*-learning (represented by CID) and *client-side*-learning (represented by ACRYL) approaches have their own advantages and disadvantages. In terms of quantitative results, the overall precision of ACRYL drops from 23.4% to 7.0%. This may suggest that ACRYL is more "noisy", i.e., it reports many non-relevant warnings. However, when considering only the *severe* warnings (Backward and Forward Bug), ACRYL achieves 28.0% precision, while CID achieves 18.4%, as highlighted in Table 2. Note that CID only reports *severe* warnings. Our results show that only 6% of non-severe warnings are fixed by developers.

In summary, the main advantages of using ***API-side*** approaches are the following:

A1: They identify more "Forward" warnings. This happens because they know which APIs disappear. Since Android APIs rarely disappear, it is more difficult to learn these rules client-side. This is why, for this category of warnings, *client-side* approaches are less effective.

A2: They are easier to keep up-to-date. Keeping up-to-date an *API-side* approach requires to run a tool every time a new version of the APIs is released. On the other hand, *client-side* approaches require a continuous monitoring of a relatively large set of apps. This operation is more resource-intensive.

Client-side approaches offer advantages as well:

C1: They provide fixing suggestions. Learning from big code-bases, *client-side* approaches allow to suggest a fix for a given compatibility issue.

C2: They support API sequences. While *API-side* approaches detect compatibility issues for a single API call at a time, *client-side* approaches can detect issues in API sequences.

In summary, there is no clear advantage in using only one approach over the other. This is particularly evident for the timeliness with which the approaches can theoretically detect issues: while *client-side* approaches can detect issues that may become bugs (as shown in the examples), they need to learn a rule for that and, therefore, many apps need to implement such a rule. On the other hand, *API-side* approaches can potentially learn a rule as soon as a

[11] https://github.com/andresth/Kandroid/commit/0b0d04
[12] https://github.com/gsantner/dandelion/commit/af0070
[13] https://github.com/no-go/RickApp/commit/05efde

new API version is released. What is evident from our results is the high complementarity of the two techniques, which points to the possibility of combining the two learning approaches in future.

Summary of RQ_1. *API-side* and *client-side* approaches are complementary. Future research may be directed at combining such approaches.

5 Study 2: Investigating the Root Causes behind Compatibility Issues

The outcome of our first study showed that: (i) there is a high complementarity between *client-side* and *API-side* data-driven approaches to detect compatibility issues, thus suggesting possible advantages coming from a hybrid approach combining both of them; and (ii) both techniques have their drawbacks, indicating the need for more advanced approaches in this area. In the first study we evaluated the effectiveness of the tools in terms of the percentage of detected issues that disappeared in time. In other words, we estimated the *precision* of the tools. In this second study, to foster research in this field, we build empirical knowledge needed to design better compatibility issue detectors. First, we define a taxonomy of root causes of compatibility issues; then, based on this information, we compute how many issues that are fixed by the developers could have been detected by the tools (*recall*): we do this to find categories of issues that can not be detected by the state-of-the-art approaches and that require the definition of new techniques.

5.1 Study Design

We manually analyze 500 pull-requests (PRs) performed by developers to address compatibility issues, with two *goals*: (i) documenting the root causes behind these issues and the fixing strategies applied by developers (i.e., how the issue has been fixed), and (ii) estimating the capability of the tools in finding the compatibility issues fixed by the developers (*recall*). In particular, we address the following research questions:

RQ_2: *What are the root causes of compatibility issues?* We want to understand what are the common causes behind compatibility issues experienced by developers in Android apps. As a result, we build a taxonomy of "compatibility issue types", and we qualitatively discuss interesting examples and fixing strategies applied by developers.

RQ_3: *To what extent can data-driven techniques detect compatibility issues fixed by developers?* We want to understand (i) if the data-driven techniques considered in our paper would have found real compatibility issues addressed in the PRs and (ii) which categories of the taxonomy we define with RQ_2 are covered and which ones are not.

5.1.1 Data Collection and Analysis

We consider the same 668 open-source projects used in our previous study (Section 4). As a first step, we extracted from the GitHub repositories of the subject apps all the PRs that were accepted and merged into the master branch before July 2019. In total, we extracted 41,381 PRs. The apps we considered targeted different API levels in their history, ranging from 1 to 29. Then, we automatically selected from this set the PRs that are likely related to the fixing of compatibility issues. To do this, we used a keyword-matching mechanism on the entire PRs, looking for PRs reporting relevant keywords. While the complete list of keywords is available in our replication package (Scalabrino et al. 2020), a few representative examples

are: *incompatib**, *compatibility issue*, *compatibility problem*, *new sdk*, *api level*, *etc.* On top of this, we also included all code-names of the existing Android versions (e.g., Nougat, Lollipop, Marshmellow). Note that the keyword-matching mechanism is likely to return false positives (i.e., PRs unrelated to the fixing of compatibility issues). For example, with our keywords matched many PRs simply reporting the text "Tested on Android Nougat" because programmers tested an application to verify the presence of an API compatibility issue. However, this is not a problem for our study since we will manually analyze the selected PRs, excluding the false positive ones. Also, while our list of keywords is certainly not exhaustive and likely to miss relevant PRs, our goal here is not to be comprehensive, but only to identify a good number of PRs to manually inspect in our study. In total, we automatically extracted 1,775 PRs. From these, we randomly sampled 500 PRs that we manually analyzed.

Note that the choice of only considering PRs, ignoring, for example, commits performed with the aim of fixing a compatibility issues but not subject of a PR, was due to two reasons. First, PRs often report the developers' discussion, that can help our manual analysis by providing additional information about (i) the compatibility issue being fixed, and (ii) the adopted solution. Second, most PRs are subject of code review activities, reducing the chance of considering in our study wrong/partial fixes.

The manual analysis was conducted by the authors of the paper. We report in Table 5 the Java and Android experience of such annotators. To support the manual analysis on the 500 PRs, we developed a web application presenting to the annotator (i) the title of the PR, (ii) the complete list of keywords matched in the PR, and (iii) the link to the PR on GitHub. Each PR was assigned to two different annotators, that were asked to report: (i) whether the PR regarded a compatibility issue (i.e., discard false positives); (ii) the *root cause* of the compatibility issue (e.g., usage of a deprecated API); and (iii) the *strategy* used to fix the issue (e.g., adding a `SDK_INT` check). We did not define a fixed set of possible values for such fields; instead, the annotators were free to define proper "tags" expressing, e.g., the root cause behind the inspected PR. To define these tags, the annotators manually inspected the PR discussion and the diff of the commits it included.

Every time the authors had to tag a PR, the web application also showed the list of tags created so far, allowing the tagger to select one of the already defined tags. In a context like the one encountered in this work, where the number of possible tags (e.g., root cause behind the compatibility issue) is extremely high, such a choice helps using consistent naming while not introducing a substantial bias. All annotators were instructed to use the *Unclear* tag in case the root cause and/or the implemented solution could not be understood.

After each PR was tagged by two annotators, 192 PRs had a conflict (i.e., a different tag used for classifying the root cause **and/or** the applied solution). These conflicts were solved through an open discussion among the involved annotators. Then, two of the authors

Table 5 Experience of the annotators who participated in Study 2

Annotator	Experience	
	Java	Android
Annotator 1	10 years	4 years
Annotator 2	15 years	3 years
Annotator 3	18 years	6 years
Annotator 4	6 years	1 years

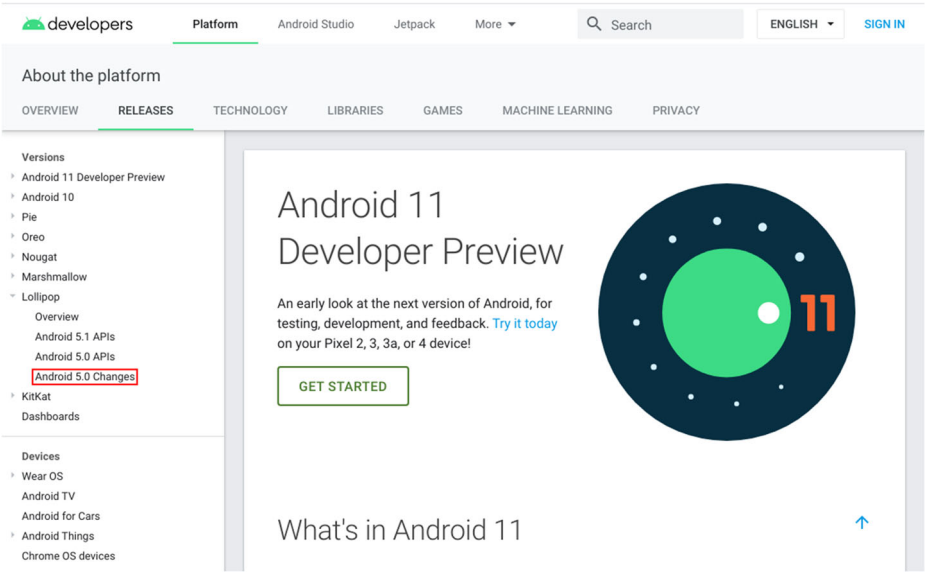


Fig. 8 Example of page considered to complement the taxonomy through the Android release notes

performed a card sorting activity (Spencer 2009) for the tags used to classify the root cause and the fixing strategy. In this phase, similar tags having the same meaning were merged and the remaining tags were grouped into categories. The result of this activity is represented by a hierarchical taxonomy of root causes behind compatibility issues (see Fig. 9). Each annotator worked about 3 hours per day, for a total of 8 days.

The PRs we analyzed may not cover all the possible causes of compatibility issues. Therefore, we also complemented our taxonomy using the Android release notes. To do this, we first extracted all the available documentation pages regarding compatibility issues: we started the analysis from the Android web page dedicated to the release notes,[14] and then we selected the sections that contained the word "change" for all the Android versions listed on the left column (e.g., "Android 5.0 changes"). Figure 8 shows an example of link we followed to access the release notes of Android 5.0. Then, two of the annotators that originally categorized the PRs independently read and categorized the release notes provided by the Android developers. At first, they tried to use the categories from the taxonomy; then, if no category suited the specific compatibility cause, they introduced new categories in the taxonomy. In total, there were 7 conflicts: they were resolved after a short discussion. We analyzed Android release notes from 5.0 (no previous release note was available) to 10.0 (the latest stable release to the date of the analysis), for a total of six major releases.

We answer RQ_2 by presenting the obtained taxonomy, discussing interesting examples and fixing strategies applied by developers for each of the main categories. The raw data, including the root cause applied in each PR as well as the identified fixing strategy, are available in our replication package (Scalabrino et al. 2020).

To answer RQ_3, we run the approaches we compared in the first study, i.e., CID (*API-side*) and ACRYL (*client-side*), both before and after the merge of the pull requests. More

[14] https://developer.android.com/preview

specifically, given a pull request P, we first find its merge commit c through the GitHub APIs. The merge commit represents the commit on the *master* branch in which the changes made on the PR branch were merged into the *master*. We also kept into account the first parent of c, $c_{\sim 1}$, i.e., the commit on the *master* branch that did not have any of the changes applied by the developers in the PR branch. We run both the tools on c and $c_{\sim 1}$. We say that an approach correctly detects a compatibility issue if at least one of the problems reported in $c_{\sim 1}$ was not reported anymore in c. We run the tools also after the merge because the method in which the tools report the warning can be different from the one modified by the developer in the fix. We report the cases in which each of the tools (ACRyL and CiD) could find the compatibility issue discussed in the PRs, i.e., the ones in which (i) at least a warning reported in the version before the merge of the PR disappeared and (ii) it was related to the fix of the issue reported in the PR. To do this, we use the same experimental design described in Section 4. In the first study we built a collection of rulesets at specific dates in the past that we used to run ACRyL at future dates. In this case, we used the same rulesets to run ACRyL: given a commit to analyze (either the one before of after the merge of a given PR), for ACRyL we used the ruleset built at the nearest date that preceded the commit time. For example, if the commit was dated 2016/02/27 and we had rulesets built at 2017/02/26 and 2017/03/01, we pick the former to avoid learning from the future. Similarly to the first study, for CiD we used as target platform the latest one available at the date of the analyzed commit. We excluded from the analysis the PRs for which (i) the merge commit of the PR could not be found in the repository (55 cases), and (ii) the build failed at either the commit before or after the merge of the PR (63 cases). In total, we could not analyze 118 PRs out of the 230 PRs we considered in our taxonomy. We report the number of issues detected by both tools for each category.

5.2 RQ_2: What are the Root Causes of Compatibility Issues?

Among the 500 PRs we analyzed, we classified 257 of them as false positives, i.e., PRs that did not actually regard compatibility issues. Of the remaining 243 PRs, 13 received the *unclear* tag, indicating that we could not understand the root cause behind the compatibility issue. Thus, our hierarchical taxonomy of root causes behind compatibility issues (Fig. 9) is based on the classification obtained for 230 PRs. We discuss the first- and second-level nodes of the taxonomy by presenting qualitative examples and reporting the fixing strategies adopted by the developers. We report in Table 6 some statistics about the apps from which we extracted the 230 PRs that regard compatibility issues. It can be noticed that they are popular in the GitHub community (∼1K stars, on average), they have a large number of contributors (∼49, on average), and they are quite big (∼24k LOCs, on average).

5.2.1 Android APIs

The vast majority of the problems fixed in the PRs we analyzed (199 PRs) were caused by changes in the Android APIs. We identified many different causes and we divided them into *Functional problems*, *GUI handling*, *API security*, *Support for new Android features*, and *Energy saving*. We discuss such categories below in details. For each category, we first provide a broad description of the cause, then we discuss some examples of PRs and we detail the Android release notes that contain references to such kind of possible issues. Finally, we provide information about the fixing strategies adopted to fix the issues: we only discuss the most common ones, without explicitly reporting details about the ones very specific to the app.

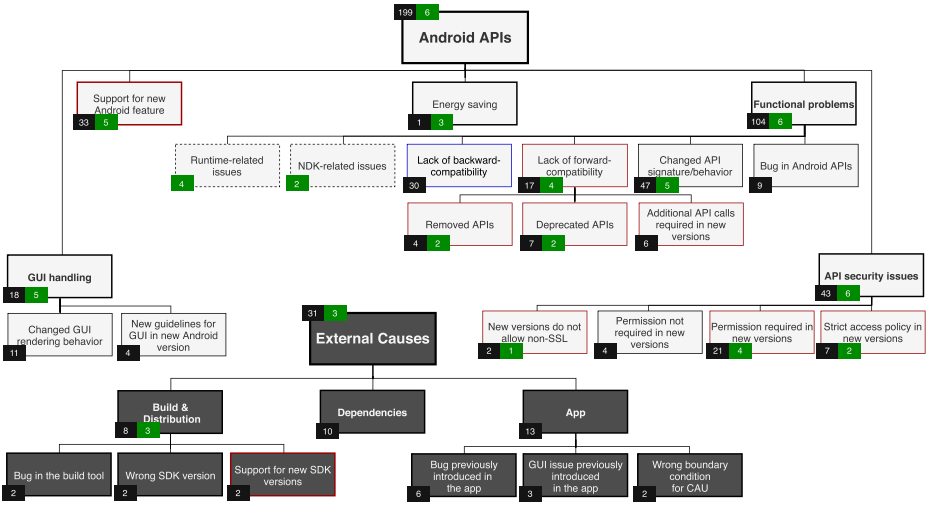


Fig. 9 Taxonomy of the causes of compatibility issues. The numbers in the black boxes represent the number of PRs we found for each category, while the numbers in green boxes represent the number of Android releases which contain category-related release notes. Dashed boxes indicate categories not found in PRs. A dark-red border indicates that the category can only cause *forward-compatibility* issues, while a dark-blue one indicates that it can only cause *backward-compatibility* issues. The others can cause both

Functional Problems We say that a compatibility issue is caused by a *functional problem* when it relates to generic changes in the Android APIs that modify some of their functional aspects (e.g., the behavior of an API changes). Most of the causes behind the compatibility issues we analyzed are functional (105 PRs). Several of these problems were caused by *backward-compatibility* issues (30 PRs): the app works well on new platform versions but, when it is executed on older versions, it presented bugs (e.g., it crashed when some actions were performed by the user). In general, we say that there is a *backward-compatibility* issue when the problem appears in older Android versions because the app uses APIs not yet available in such versions (e.g., the API was introduced in version 26 and the app supports version 25). The most common cause is that some used APIs were simply not available in older SDK versions. For example, PRn4897 of the app Anki Android[15] fixed a bug occurred because the method `setContentDescription` of the class `android.widget.RemoteViews` was not available for the SDK versions below 15. For this reason, the authors added a CAU to call such a method only when the app was executed on more recent Android versions. We found two additional functional categories by analyzing the Android release notes: *Runtime-related issues* and *NDK-related issues*. The first one refers to possible issues due to major changes in the runtime environment in which the apps are executed. For example, in version 5.0 the old runtime environment, Dalvik, was replaced by ART (Android Runtime). Besides, in version 6.0,[16] a possibly breaking change was done to ART: "On previous versions of Android, if your app requested the system to load a shared library with text relocations, the system displayed a warning but still allowed the library to be loaded. Beginning in this release, the system rejects this library if your app's target SDK version is 23 or higher.". *NDK-related issues*, instead, regard the

[15] https://github.com/ankidroid/Anki-Android/pull/4897

[16] https://developer.android.com/about/versions/marshmallow/android-6.0-changes

Table 6 Statistics about 20 out of 94 apps considered in Study 2, i.e., the ones with the highest number of PRs. We only report aggregate statistics for the others for space limitations. The complete list is available in the replication package (Scalabrino et al. 2020)

Application	PRs	Stars	Contributors	LOCs
AnkiDroid	26	2,442	161	51,466
Seafile Android Client	10	370	33	33,914
KISS	9	1,414	135	10,698
Amaze File Manager	8	2,823	105	39,223
Tusky	8	832	113	28,551
AntennaPod	7	2,717	122	51,323
syncthing-android	7	1,297	42	9,031
NewPipe	7	7,813	323	37,129
DuckDuckGo Android	6	1,172	27	31,741
K-9 Mail	6	4,831	210	102,256
Wikimedia Commons Android	5	562	199	28,075
Tachiyomi	5	6,458	71	28,365
Nextcloud Android	5	1,700	115	72,420
wallabag	5	298	65	14,165
LeafPic	4	2,998	33	14,379
Open Food Facts	4	436	86	30,586
Silence	4	1,022	134	41,534
Riot-Android	4	1,285	235	66,418
Kore - Kodi/XBMXC remote for Android	4	379	75	37,834
Slide	3	1,185	71	79,051
Others (average)	1.4	720	39	19,039
Total average	2.6	1,014	49	23,634

Native Development Kit that may be used by developers to include native components in their apps. NDK may help developers reuse libraries written in other languages (e.g., C++) and may improve the performance of some functionalities. An example of NDK-related change that may introduce compatibility issues is provided in the release notes of Android 7.0[17]: "Starting in Android 7.0, the system prevents apps from dynamically linking against non-NDK libraries, which may cause your app to crash." We could not find any issues in the PRs we analyzed regarding these two categories of issues. This is probably because (i) runtime-related issues are rare, and (ii) few apps use Android NDKs.

A lower — but still substantial — number of problems we analyzed were caused by *forward-compatibility* issues (17 PRs): the app worked with older SDK versions but, when a new SDK version was introduced and it was targeted by the app, some bugs manifested. In general, we say that there is a *forward-compatibility* issue when the problem appears in newer Android versions because old APIs not available anymore (e.g., the API existed up to version 25, then it was removed, and the app targets version 26). In seven cases, deprecated APIs were replaced by developers with new APIs to avoid the bug manifestation. In some other cases, additional API calls were necessary to use Android features that were available

[17] https://developer.android.com/about/versions/nougat/android-7.0-changes

also in older Android versions (6 PRs). Finally, we found four cases in which some APIs were removed in new SDK versions and, therefore, this resulted in a crash; for example, in PRn45 of Silence IM,[18] the developers needed to define both a normal and a legacy MMS handler, since some APIs for MMS handling were modified since Android 5.1, together with other carrier services.[19]

The release notes of four out of six Android versions (5.0, 6.0, 9.0, and 10.0) mention the deprecation or the removal of some APIs. For example, in the release notes of Android 6.0 it can be found that such a version "removes support for the `Apache HTTP` client". The Android developers suggest: "If your app is using this client and targets Android 2.3 (API level 9) or higher, use the `HttpURLConnection` class instead. This API is more efficient because it reduces network use through transparent compression and response caching, and minimizes power consumption."[16]

Besides such two macro-categories, one of the most common functional causes of compatibility issues was a change in API signature and/or behavior (47 PRs): the features available were the same in different Android versions, but either the signature or the behavior of the API changed. PRn160 of the andOTP app is an interesting example of this[20]: the app asks a PIN and it allows the user to generate OTPs (One Time Passwords) only if the PIN is correct. However, on more recent Android versions, the app crashed if no PIN was inserted and it restarted allowing the user to generate OTPs. This problem was caused by the fact that an `IllegalArgumentException` was thrown in newer API versions, and it was not handled by the app, indirectly introducing a security problem. Almost all the Android release notes we analyzed introduced some changes in the behavior of the APIs. The only exception is represented by the version 7.0, which introduced fewer compatibility-related changes with respect to the others. For example, the release notes of Android 8.0 state: "The `getSaveFormData()` method now returns `false`. Previously, this method returned `true` instead.". Any application relying on the return value of such a method needed to handle this change.

Not surprisingly, our results show that only a minority of compatibility issues were caused by bugs in Android APIs (9 PRs). For example, PR #731 of Tusky app added a workaround for a bug present in Android 8[21]; interestingly, a developer commented: "*Should a similar workaround be applied to any other text editors?*". This suggests that ACRyL would have been useful for the developers since it would have allowed them to detect and fix other possible compatibility issues in the app.

To fix the issues that have a functional impact on the app, the developers, in most of the cases (42 PRs), add or update a check to `SDK_INT` and they specify different behaviors based on the current platform (i.e., they implement a CAU). This confirms that analyzing CAUs is useful to acquire knowledge about wrong API usages and fixing patterns. Some of those PRs also use annotations such as `@TargetApi` in combination with CAUs. We also found that, for a non-negligible amount of PRs (31), the developers fix the issue without implementing a CAU, but simply using a generic fix.

Lesson 1. CAUs are the main source of information that can be used by data-driven detection techniques. However, other information, such as annotations, could be exploited to develop better detection tools.

[18] https://github.com/SilenceIM/Silence/pull/45

[19] https://developer.android.com/about/versions/android-5.1

[20] https://github.com/andOTP/andOTP/pull/160

[21] https://github.com/tuskyapp/Tusky/pull/731

API Security A compatibility issue is caused by *API security* aspects when it is related to changes in security aspects in the Android platform (e.g., permission handling). This is the second most common cause of compatibility issues (43 PRs). Most of the issues were caused by problems with the Android permission handling (25 PRs): in some cases, new permissions were required to access some resources (21 PRs); in some others, permissions were not required anymore (4 PRs). For example, in PR #76 of Tickmate,[22] the developers only required a permission for accessing external storages for older SDK versions, since this permission was not required anymore in newer SDK versions.

Some issues were caused by a stricter Android access policy to resources (7 PRs). It is worth noting that while the previously described issues could be fixed by simply adding/removing permissions requests, such issues were harder to fix, because Android did not allow the access to some resources anymore. For example, in PRn15 of the app PortKnocker,[23] the developers needed to stop using the URI schema "file:" since Android did not allow this anymore in newer versions.

Finally, we found two cases in which the problem was caused by the fact that new SDK versions did not allow non-SSL encrypted web traffic. For example, in PR #3053 of Firefox Focus[24] the developers incidentally noticed that, after targeting a new Android SDK version, the browser was not allowed the access non-HTTPS web pages anymore. To fix this problem, the developers needed to explicitly disable this new Android security feature.

The Android release notes often reported security-related changes that could be the cause of compatibility issues. Most of such changes involved permissions (5.0, 6.0, 9.0, and 10.0). For example, Android 6.0 introduced runtime permission checks: "On your apps that target Android 6.0 (API level 23) or higher, make sure to check for and request permissions at runtime. [...] Even if your app is not targeting Android 6.0 (API level 23), you should test your app under the new permissions model."[25]. Besides permissions, many changes involved other generic security issues, such as the deprecation of insecure cryptographic algorithms. Many changes also regarded privacy-specific issues: for example, in the release notes of Android 8.0 it can be found: "In Android 8.0 (API level 26) and higher, queries for usage data return approximations rather than exact values. The Android system maintains the exact values internally, so this change does not affect the auto-complete API."[26] Apps that assume that such values are exact rather than approximated may encounter functional compatibility problems.

To fix issues involving API security, also in this case the developers mostly used CAUs (14 PRs). As expected, in this case, many fixing strategies involve changes to permissions requests/checks (9 PRs).

> **Lesson 2.** To improve the performance of compatibility issues detection tools, security-specific checks should be implemented. Above all, it could be possible to mine information about permissions required to use APIs for different SDK versions (see, e.g., Backes et al. (2016) and Bartel et al. (2012)).

GUI Handling We say that a compatibility issue is induced by such a category of causes when GUI handling changes in different versions of the Android platform and some GUIs

[22] https://github.com/lordi/tickmate/pull/76
[23] https://github.com/xargsgrep/PortKnocker/pull/15
[24] https://github.com/mozilla-mobile/focus-android/pull/3053
[25] https://developer.android.com/about/versions/marshmallow/android-6.0-changes
[26] https://developer.android.com/about/versions/oreo/android-8.0-changes

of a given app are not working as intended because of this. We found 18 PRs for which the cause could be traced to such a category. In some cases (11 PRs), the issue was caused by a difference in the GUI rendering. For example, in PRn286 of QuasselDroid[27], the developers had to change the text style assigned through a `span` markup object since the previously used one made the text invisible in new versions of the SDK. In some other cases (4 PRs), new SDK versions provided different policies for handling some specific GUI components. Finally, we found cases in which GUI properties were deprecated (1 PR) or in which GUI components were not available in older Android versions (1 PR). Compatibility issues can arise also when the behavior of the Android GUI handling system is modified. For example, in OpenHAB, the developers heavily relied on the internal hierarchy of GUI components to implement a feature. Since such a hierarchy changed unexpectedly in newer Android versions, in PR #592[28] the developers needed to fix the resulting bug.

The release notes of most Android versions (except for version 6.0) include minor changes to GUI-related elements that may cause compatibility issues. Such issues may be minor (e.g., less aesthetically appealing elements due to the usage of wrong colors) or even functional issues: for example, after 9.0,[29] "Views with 0 area (either a width or a height is 0) are no longer focusable." Apps relying on the the fact that views with 0 area can receive focus may contain bugs since such a version.

The fixing of issues involving the GUI required, in several cases (7 PRs), a simple change to the XML files describing the GUI. In other words, in many cases it is not necessary to define different strategies for different Android versions. On the other hand, we found a good amount of cases (6 PRs) that implement CAUs to fix GUI issues.

> **Lesson 3.** Detection of GUI-related compatibility issues is challenging due to the fact that, in most of cases, a generic fix is needed, and we did not observe any significant pattern characterizing these bugs. Learning from CAUs can only help in a minority of the cases.

Support for New Android Features We say that a compatibility issue is caused by a new Android feature if the introduction of such a feature affects some functionalities of the app. We found 33 PRs aimed at fixing such kind of issues. This happens mostly when new features are introduced in previously existing parts of the system. For example, the Android notification system was changed in Android 8: the notification channels were introduced to allow users ignoring entire groups of notifications. Some apps targeting Android 8 and above which did not adapt to notification channels stopped working as intended. An example can be found in the PRn794 of Wallbag[30]: when the developers updated the target SDK version of their app, notifications stopped working for Android 8 and above. To fix this issue, the developers had to introduce notification channels. The problem experienced in this app was probably related to the way the older notification APIs were used, but the developers did not find the root cause of the problem.

The release notes of most Android versions explicitly mention the new features provided to the users that are accessible to the developers through some APIs. For example,

[27] https://github.com/sandsmark/QuasselDroid/pull/286
[28] https://github.com/openhab/openhab-android/pull/592
[29] https://developer.android.com/about/versions/pie/android-9.0-changes-28
[30] https://github.com/wallabag/android-app/pull/794

in Android 9 it was introduced a feature that allows apps to enumerate cameras: this happened mostly because smartphones with two or more rear cameras arrived on the marked. The only version that does not include such changes is Android 7.

To fix such issues, developers introduce CAUs in most of the cases (7 PRs), but they also often update the app Manifest, e.g., they change the target SDK version (6 PRs) or they apply a generic fix which does not depend on the platform (6 PRs).

> **Lesson 4.** New Android features, mostly the ones that involve previously existing systems, can introduce compatibility issues. Detecting such issues is particularly difficult even for developers.

Energy Saving Compatibility issue can be caused by changes in the *energy saving* policies adopted by Android, unexpectedly affect some functionalities of the apps. We found a single PR that addressed an issue with energy consumption, i.e., PRn311 of Wallabag.[31] In such a PR, the developers needed to use a `JobScheduler` to avoid battery usage warnings in newer Android versions. Such a problem was fixed by introducing a check on the `SDK_INT`. Android release notes mentioned possible battery-related issues for 3 Android releases (i.e., from 6.0 to 8.0). Indeed, Android 6.0 introduced the Doze mode, which forcefully pauses the execution of some apps. Such a feature may result in collateral functional problems with the apps that are stopped/paused by the operating system: "This release introduces new power-saving optimizations for idle devices and apps. These features affect all apps so make sure to test your apps in these new modes"[32].

Nevertheless, we found references to possible compatibility issues due to the changed energy saving policies in the release notes of three Android releases, i.e., 6.0, 7.0, and 8.0. Energy saving can have an effect on the functionality of Android apps: new platform versions could change the policies so that some operations (e.g., the ones requiring the usage of the Bluetooth sensor) are temporarily suspended in some occasions. Apps may stop working due to such changes. It is possible that the low quantity of PRs we found belonging to such a category is due to the fact that such issues are hard to detect, reproduce and fix. Another possibility is that only few categories of apps may be affected by such changes (e.g., the ones that require a constant connection to IoT devices).

> **Lesson 5.** While the Android release notes highlight that changes in energy saving policies can have an impact on some apps, we found only one PR addressing such a kind of problem. It is unclear how such issues are fixed and how automated approaches may detect them.

5.2.2 External Causes

While most of the compatibility issues are caused by changes in the Android APIs, we found cases in which they had different causes (31 PRs). The causes could be classified into three categories: those related to the *App*, to the *Dependencies*, and to the *Build & Distribution* of the app.

App Compatibility issues are caused by the *app* when they are not related to changes in the Android APIs, but rather to errors introduced in the past. While the root causes of such

[31] https://github.com/wallabag/android-app/pull/311

[32] https://developer.android.com/about/versions/marshmallow/android-6.0-changes

issues are clearly still related to changes in the Android APIs, we use such a category for the cases in which the compatibility issue was related to bad practices in the development of the apps themselves (e.g., usage of a deprecated method that is later removed from the Android APIs). We found 13 PRs that addressed such kind of issues. In six cases, these were related to functional bugs, while others involved GUI issues (3 PRs), security problems (1 PR) and poor code quality (1 PR). We also found two cases in which the condition of the CAU was wrong. For example, in the Syncthing app, the developers used a wrong CAU check (i.e., `> Build.VERSION_CODES.N`); in PR #918,[33] they fixed the bug by simply changing the condition (`>= Build.VERSION_CODES.N`). It is worth noting that not all the updates to the CAU condition imply an error in the app: such changes may be done, for example, when the developers want to support new Android platforms (e.g., when such changes are done together with an update to the `minSDKVersion`).

The distribution of fixing strategies adopted for this kind of issues is in line with the general trend: most of the issues were fixed by adding or modifying existing CAUs (7 PRs), while fewer cases required a generic fix (3 PRs).

> **Lesson 6.** There is a risk that data-driven techniques learn sub-optimal or wrong fixing patterns from apps. However, the number of such problems is very limited ($< 6\%$). Also, a confidence level-based strategy such as the one adopted in ACRyL to discard detection rules could help in limiting this risk.

Dependencies Like bugs in general, also compatibility issues may be caused by problems in the dependencies of the app. We found 10 PRs addressing such issues. The most frequent cause is the presence of a compatibility issue in a library used by the app (8 PRs).

As expected, the main way to fix such kind of issues is to update the dependencies (6 PRs). Also, in two PRs it was required a complete replacement of a dependency, i.e., using different libraries that supported newer SDK versions.

> **Lesson 7.** A data-driven technique could learn which versions of which libraries are affected by compatibility issues and warn developers if problematic dependencies are included in their app. This aspect is completely ignored by state-of-the-art approaches.

Build & Distribution A minority of the compatibility issues in the PRs we analyzed were caused by build issues (e.g., a bug in the build tool) or by problems related to the SDK version selected by the developers. We found only 8 of such issues. For example, we found a case in which the developers needed to update the SDK version because the Play Store required them to do so to keep the app available.

The fixing strategies for such issues were diverse. The two bugs in the build tool regarded a problem with AAPT2 and, therefore, the solution was to disable such a tool. On the other hand, the other issues required an increase or decrease of the SDK minimum/target version (3 PRs) and 2 PRs required the introduction of CAUs.

Surprisingly, even if such categories are related to external causes, we found three Android releases (6.0, 9.0, and 10.0) which introduced changes that may cause issues related to "Build & Distribution". For example, Android 6.0[16] "performs stricter validation of APKs. An APK is considered corrupt if a file is declared in the manifest but not present in

[33] https://github.com/syncthing/syncthing-android/pull/918

the APK itself. An APK must be re-signed if any of the contents are removed." Therefore, the same APK may be considered corrupt by Android 6.0 and valid by previous versions: a wrong build configuration may produce an APK with compatibility issues because of this change.

> **Lesson 8.** Compatibility issues related to Build & Distribution are rarer and are generally easier to detect (e.g., the APK does not work at all on some versions). Data-driven solutions aimed at analyzing the build configuration may be introduced to find such problems.

5.3 *RQ*$_3$: To What Extent Can the Tools Find the Compatibility Issues that Developers Fixed?

Table 7 shows the results obtained by ACRyL, CiD, and both the tools combined. The results show a substantial tie between the two approaches: while ACRyL could capture more warnings belonging to the Android APIs category, CiD captured more issues caused by external factors. In all the cases, the two tools could mostly capture warnings that were fixed by adding a check on the SDK_INT. The only exception is represented by PRn221 of the app `OpenRedmine` (GH: indication/OpenRedmine): in this case, the issue was fixed by replacing a dependency. In the dependency itself, however, the problem was still tied to the presence of a missing or wrong check on the SDK_INT.

We found that existing data-driven solutions are able to capture only a portion of all the issues that developers fixed in the evolution of the apps: even combining the two approaches, the total estimated recall achieved is only ∼9% (6 out of 112 by ACRyL, 7 out 112 by CiD, and 9 out of 112 when combined). Even if such a percentage is low in general, there are differences depending on the root cause of the compatibility issue. The available approaches can not detect problems related to "Support for new Android feature" and "GUI handling". Indeed, there are some limitations related to these approaches that do not allow them to identify such warnings. We discuss below some examples for each category and we provide possible indications on how future work may address such issues. Also, it is worth

Table 7 Categories of issues captured by ACRyL, CiD, and both the tools combined

	Category	Valid PRs	ACRyL	CiD	Combined
Android					
	Functional problems	49	3	3	5
	API security issues	25	2	1	2
	Support for new Android feature	15	0	0	0
	GUI handling	7	0	0	0
	Energy saving	1	0	0	0
Ext.					
	Build & Distribution	6	0	1	1
	App	5	1	1	1
	Dependencies	4	0	1	1
	Total	112	6	7	10

noting that the tools could not detect the single issue fixed in the PR related to the category "Energy saving": we do not discuss this because it is not possible to draw generic conclusions based on a single instance and there is no inherent motivation for which such a category is "impossible" to cover for state-of-the-art approaches, differently from the others we discuss.

Support for New Android Feature It was shown in Section 5.2 that even developers struggle finding and fixing such issues. Finding such issues may be very challenging for *API-side* approaches: while they may be able to find backward-compatibility issues related to such problems (e.g., notification channels used before they were introduced), they would not be able to find forward-compatibility issues (e.g., notification channels not used after API level 25): if an API is available since a specific version, it rarely means that all the apps should use it. Instead, a *client-side* approach able to find issues related to new Android features may be defined: CAUs similar to ($\leq X$, {}, `newFeature`) can be found in client apps when they start implementing the new feature, where X indicates the version from which the new feature is available and `newFeature` indicates the API sequence needed in the new version: this happens when client apps want to support older Android platforms. If many clients use such CAUs, there is a possibility that such feature should be used. For example, the CAU ($\leq$ 25, {}, {`NotificationManager.createNotificationChannel`, `NotificationChannel.` < `init` >}) is adopted by 4 apps at the latest date we analyzed in our study. At the moment, ACRyL uses such a kind of CAUs only to detect backward-compatibility issues, just like an *API-side* would do. Future work aimed at experimenting the use of such CAUs for detecting such problems may be done.

GUI Handling Issues related to the graphical user interface are rarely fixed using CAUs: the currently available *API-side* and *client-side* approaches would help finding such problems only in a minority of cases, since they only take into account the source code. Finding such issues would require the definition of new approaches that also target the XML files used for defining GUIs in Android. For example, it would be possible finding patterns of XML elements that result in compatibility issues through a new *client-side* approach.

Summary of RQ_3. The recall of the state-of-the-art tools, even when combined, is lower than 9%. Issues related to some root causes (specifically, "Support for new Android feature" and "GUI handling") are harder or even impossible to detect for state-of-the-art approaches. New approaches may be defined to detect such problems.

6 Threats to Validity

Threats to *construct validity* relate to possible measurement imprecision when extracting data used in our study. When identifying the fixed API compatibility issues in Study I, we assume that a compatibility issue identified in an app's snapshot S_i by ACRyL (CID) and not detected anymore by the same tool in a subsequent snapshot has been intentionally fixed by developers. It may happen that developers stop use an API causing the compatibility issue not due to this latter, but just because this API is not needed anymore in the app's code.

Despite this, we applied strict pre-/post-conditions to at least limit the *false positive* fixing instances (see Section 4). For example, we do not consider an issue as fixed if the method affecting it was deleted in a subsequent snapshot.

To make the comparison between ACRyL and CiD fair we (i) used the original CiD implementation as provided by the tool's authors, and (ii) only compared the compatibility issues identified by the two tools on the set of apps on which both tools correctly worked.

In the second study, the automatic mining of PRs based on keywords-matching mechanisms resulted in the retrieval of some false positives. These imprecisions were discarded during our manual analysis, thus they did not affect our findings.

Also, in our manual analysis, we based the classification of the root causes on what was visible in the PR discussion and in the diff of its related commits. It is possible that the analyzed information is incomplete, for example due to the fact that an issue was partially discussed by developers via chat.

Threats to *internal validity* concern confounding factors, internal to our study, that can affect the results. In Study I, we performed the calibration of the ACRyL's parameter on snapshots belonging to a time interval not used in our study. The time needed to fix an issue reported in Fig. 7 can contain random errors, because we observe only some snapshots of the apps. For example, if an issue is introduced on date X and fixed on date $X + 20$ but we only keep into account snapshot on dates $X + 10$ and $X + 20$, the time needed to fix is underestimated (10 days instead of 20). The opposite could have occured as well.

In Study II, threats in this category are related to possible subjectiveness introduced during the manual analysis. We mitigated this threat by making sure that each PR was independently analyzed by two authors and that conflicts were solved via an open discussion.

In our second study we considered only apps available on the F-droid store (i.e., a subset of the apps considered in our first study). We report in Table 6 some statistics about such apps: the average number of stars, contributors and LOC are quite high ($\sim$1k, $\sim$49, and $\sim$24k, respectively). Therefore, we can assume that the quality of the apps we considered is good. We only considered two apps with less than 10 stars: FRCAndroidWidget (1 PR) and gift-card-guard (2 PRs).

Threats to *external validity* represent the ability to generalize the observations in our study. Study I is performed on a set of 11,863 snapshots from 585 apps. The main issue is therefore related to the fact that all used apps are open source, and might not be representative of commercial apps. Concerning Study II, it is possible that our taxonomy of root causes depends on the particular set of PRs we analyzed, and that in other contexts developers fix compatibility issues we did not encounter. Also, for availability reasons, we only focused on open-source applications for both our studies. Specifically, for Study I we used F-Droid and for the Study II we used the source code of some applications hosted on GitHub. We could not use closed-source apps for our studies: as for Study I, the presence of obfuscated code or the generated APKs would not allow us to track the warnings among different snapshots (the classes/methods may have different names); as for Study II, we do not have access to any non open-source repository providing PRs that we could analyze. As for the first study, it is possible that the precision of the two compared tools changes when they are used on closed-source apps. As for the second study, we complemented the taxonomy defined through the PRs with the information available in the Android release notes. Therefore, we believe that such a taxonomy is quite generic and also valid for closed-source apps.

7 Conclusion and Future Work

Android fragmentation forces developers to support many versions of the OS to increase their potential market share. However, the evolution of Android APIs can make such a task harder, because it increases the effort in testing and the risks of introducing bugs only reproducible in some versions.

We compared in a large empirical study two different types of data-driven approaches, *API-side* (represented by CID, the state of the art) and *client-side* (represented by ACRYL, our new tool), both aimed at detecting compatibility issues early.

The results show that the two strategies are complementary. The comparison shows no clear winner as they both have their own advantages and disadvantages. Also, the performance ensured by the two tools clearly pointed to the need for more research in this field, with the goal of building better approaches for detecting compatibility issues (e.g., by combining both techniques in a hybrid approach).

To provide hints for a research agenda in this field and to understand possible drawbacks of the approaches defined in the state-of-the-art, we run a qualitative study in which we manually analyzed 500 pull requests likely related to the fixing of compatibility issues. After discarding 255 false positive instances, we (i) built a taxonomy of compatibility issues faced by Android app developers, and (ii) estimated the *recall* of the apporaches by checking if the tools would have been able to detect the issues fixed in the PRs. We discussed the categories of compatibility issues we identified, using the acquired empirical knowledge to draw a number of lessons learned potentially useful for the building of better detection tools. Finally, we found that state-of-the-art tools can only detect less than 10% of the issues fixed by developers, with some categories of the taxonomy ("Support for new Android feature", "GUI handling" , and "Energy saving") left completely uncovered.

Our future work includes the definition of a hybrid approach which uses both *API-side* and *client-side* information to provide a higher variety of compatibility issues and to mitigate the limitations of both strategies. Besides, as we showed in Section 5.2, developers would benefit from approaches specifically aimed at automatically detecting GUI-related and security-related issues. Finally, future *client-side* approaches should consider other sources of information besides CAUs to learn rules and detect issues.

References

Amann S, Nadi S, Nguyen HA, Nguyen TN, Mezini M (2016) MUBench: A benchmark for API-misuse detectors. In: Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories, MSR, pp 464–467. https://doi.org/10.1109/MSR.2016.055

Amann S, Nguyen HA, Nadi S, Nguyen TN, Mezini M (2018) A systematic evaluation of static API-misuse detectors. IEEE Transactions on Software Engineering, https://doi.org/10.1109/TSE.2018.2827384

Backes M, Bugiel S, Derr E, McDaniel P, Octeau D, Weisgerber S (2016) On demystifying the android application framework: Re-visiting android permission specification analysis. In: 25th {USENIX} security symposium ({USENIX} security 16), pp 1101–1118

Bartel A, Klein J, Le Traon Y, Monperrus M (2012) Automatically securing permission-based software by reducing the attack surface: An application to android. In: 2012 Proceedings of the 27th IEEE/ACM international conference on automated software engineering. IEEE, pp 274–277

Bavota G, Linares-Vásquez M, Bernal-Cárdenas CE, Penta MD, Oliveto R, Poshyvanyk D (2015) The impact of API change- and fault-proneness on the user ratings of Android apps. IEEE Trans Softw Eng 41(4):384–407. https://doi.org/10.1109/TSE.2014.2367027

Brito G, Hora A, Valente MT, Robbes R (2016) Do developers deprecate APIs with replacement messages? A large-scale analysis on java systems. In: Proceedings of the 23rd IEEE International

Conference on Software Analysis, Evolution, and Reengineering, SANER, vol 1, pp 360–369. https://doi.org/10.1109/SANER.2016.99
Choudhary SR, Gorla A, Orso A (2015) Automated test input generation for Android: Are we there yet? In: Proceedings of the 30th IEEE/ACM international conference on automated software engineering, IEEE Computer Society, ASE, pp 429–440, https://doi.org/10.1109/ASE.2015.89
Dilhara M, Cai H, Jenkins J (2018) Automated detection and repair of incompatible uses of runtime permissions in android apps. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems. ACM, pp 67–71
Fazzini M, Orso A (2017) Automated cross-platform inconsistency detection for mobile apps. In: Proceedings of the 32Nd IEEE/ACM international conference on automated software engineering. IEEE Press, pp 308–318
Han D, Zhang C, Fan X, Hindle A, Wong K, Stroulia E (2012) Understanding android fragmentation with topic analysis of vendor-specific bugs. In: Proceedings of the 19th working conference on reverse engineering. WCRE, pp 83–92 https://doi.org/10.1109/WCRE.2012.18
He D, Li L, Wang L, Zheng H, Li G, Xue J (2018) Understanding and detecting evolution-induced compatibility issues in Android apps. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. ACM, ASE, pp 167–177, https://doi.org/10.1145/3238147.3238185
Joorabchi ME, Mesbah A, Kruchten P (2013) Real challenges in mobile app development. In: Proceedings of the ACM/IEEE International symposium on empirical software engineering and measurement. ESEM, pp 15–24 https://doi.org/10.1109/ESEM.2013.9
Li L, Bissyandé TF, Le Traon Y, Klein J (2016) Accessing inaccessible Android APIs: An empirical study. In: Proceedings of the IEEE international conference on software maintenance and evolution. ICSME, pp 411–422 https://doi.org/10.1109/ICSME.2016.35
Li L, Bissyandé TF, Wang H, Klein J (2018a) 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. ISSTA, pp 153–163
Li L, Gao J, Bissyandé TF, Ma L, Xia X, Klein J (2018b) Characterising deprecated Android APIs. In: Proceedings of the 15th international conference on mining software repositories. MSR, pp 254–264
Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyvanyk D (2013) API change and fault proneness: A threat to the success of Android apps. In: Proceedings of the 9th Joint meeting on foundations of software engineering. ACM, ESEC/FSE, pp 477–487, https://doi.org/10.1145/2491411.2491428
Linares-Vásquez M, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D (2014) How do API changes trigger stack overflow discussions? a study on the Android SDK. In: Proceedings of the 22nd International Conference on Program Comprehension. ACM, ICPC, pp 83–94, https://doi.org/10.1145/2597008.2597155
Linares-Vásquez M, Moran K, Poshyvanyk D (2017) Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution. ICSME, pp 399–410 https://doi.org/10.1109/ICSME.2017.27
Luo T, Wu J, Yang M, Zhao S, Wu Y, Wang Y (2018) MAD-API: Detection, Correction and explanation of API misuses in distributed android applications. In: Proceedings of the 7th International conference on artificial intelligence and mobile services. Springer International Publishing, pp 123–140
McDonnell T, Ray B, Kim M (2013) An empirical study of API stability and adoption in the Android ecosystem. In: Proceedings of the IEEE international conference on software maintenance. IEEE Computer Society, ICSM, pp 70–79. https://doi.org/10.1109/ICSM.2013.18
Mutchler P, Safaei Y, Doupé A, Mitchell J (2016) Target fragmentation in Android apps. In: Proceedings of the IEEE Security and Privacy Workshops, SPW, pp 204–213, https://doi.org/10.1109/SPW.2016.31
Robbes R, Lungu M, Röthlisberger D (2012) How do developers react to API deprecation?: The case of a Smalltalk ecosystem. In: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. ACM, FSE, pp 56:1–56:11 https://doi.org/10.1145/2393596.2393662
Sawant AA, Robbes R, Bacchelli A (2016) On the reaction to deprecation of 25,357 clients of 4+1 popular java APIs. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME, pp 400–410 https://doi.org/10.1109/ICSME.2016.64
Scalabrino S, Bavota G, Linares-Vásquez M, Lanza M, Oliveto R (2019) Data-driven solutions to detect API compatibility issues in android: an empirical study. In: Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26–27 May 2019, Montreal, Canada pp 288–298
Scalabrino S, Bavota G, Linares-Vásquez M, Piantadosi V, Lanza M, Oliveto R (2020) Replication package. https://dibt.unimol.it/report/acryl-emse/
Spencer D (2009) Card sorting: Designing usable categories. Rosenfeld Media

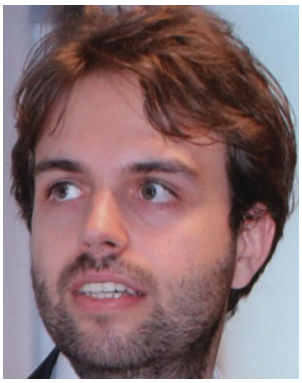
Wei L, Liu Y, Cheung SC (2016) Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE, pp 226–237

Wu D, Liu X, Xu J, Lo D, Gao D (2017) Measuring the declared SDK versions and their consistency with API calls in Android apps. In: Proceedings of the International Conference on Wireless Algorithms, Systems, and Applications. Springer International Publishing, pp 678–690

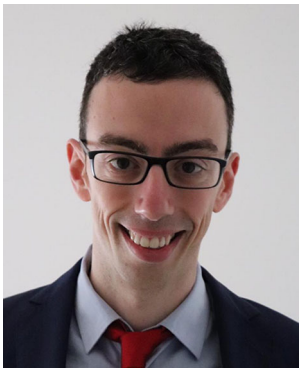
Zhang Z, Cai H (2019) A look into developer intentions for app compatibility in android. In: 2019 IEEE/ACM 6th international conference on mobile software engineering and systems, MOBILESoft. IEEE, pp 40–44

Zhou J, Walker RJ (2016) API Deprecation: a retrospective analysis and detection method for code examples on the web. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, FSE, pp 266-277, https://doi.org/10.1145/2950290.2950298

Zhou X, Lee Y, Zhang N, Naveed M, Wang X (2014) The peril of fragmentation: Security hazards in Android device driver customizations. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE Computer Society, SP, pp 409–423, https://doi.org/10.1109/SP.2014.33



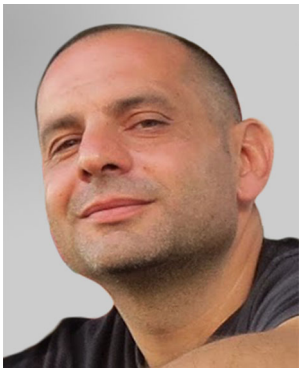
Simone Scalabrino is a Postdoc researcher at the University of Molise, Italy. He received his Ph.D. from the University of Molise in 2019, defending a thesis on assessing and improving source code understandability and readability. He received his Master's Degree in Computer Science from the University of Salerno in 2015. His research interests include software quality, testing and security. He received three ACM SIGSOFT Distinguished Paper awards at ICPC 2016, ASE 2017, and MSR 2019. He served as Local Arrangement Co-Chair for SANER 2018. He is co-founder and CSO of *datasound*, a spin-off of the University of Molise.



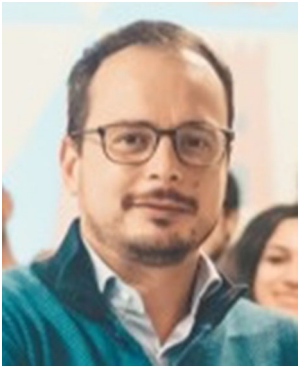
Gabriele Bavota is an Associate Professor at the Universitá della Svizzera italiana (USI), Switzerland. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. His research interests include software maintenance, empirical software engineering, and mining software repository. He is the author of over 120 papers appeared in international journals, conferences and workshops. He served as a Program Co-Chair for ICPC'16, SCAM'16, and SANER'17. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, FSE, ASE, ICSME, MSR, SANER, ICPC, SCAM, and others.

Mario Linares-Vásquez is an Assistant Professor at Universidad de los Andes in Colombia. He received his Ph.D. degree in Computer Science from the College of William and Mary in 2016. He received his B.S. in Systems Engineering from Universidad Nacional de Colombia in 2005, and his M.S. in Systems Engineering and Computing from Universidad Nacional de Colombia in 2009. He is the leader of The Software Design Lab, which is focused on automated software engineering, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks, design for everyone, and app development with societal impact. He received four ACM SIGSOFT distinguished paper awards, and the best paper award at ICSM'13. He has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, ASE, ICSME, MSR, SANER, ICPC, SCAM, MOBILESOFT and others. Mario is member of the editorial board of the Journal of Systems and Software and the Information and Software Technology Journal.

Valentina Piantadosi received (magna cum laude) a Master's Degree in Software System Security from the University of Molise (Italy) in 2018 defending a thesis on Software Reliability and Testing, advided by Prof. Rocco Oliveto. She received a Bachelor's Degree in Computer Science from the University of Molise in 2016 defending a thesis on Software Refactoring, advised by Prof. Rocco Oliveto. She is currently a Ph.D. Student at the Department of Bioscences and Territory of University of Molise, advised by Prof. Rocco Oliveto. Her research interests include Vulnerability Detection, Testing and Machine Learning.



Michele Lanza is full professor and director of the Software Institute at the Universit della Svizzera italiana (USI) in Lugano, Switzerland. His doctoral dissertation, completed in 2003 at the University of Bern in Switzerland, received the Ernst Denert Award for best thesis in software engineering of 2003. Prof. Lanza received the Credit Suisse Award for best teaching in 2007 and 2009. He has graduated 12 awesome PhD students so far, some of which have become internationally well-known academics. At USI Prof. Lanza directs the Software Institute of USI, founded in 2017. Moreover, he leads the REVEAL research group, which he founded in 2004. REVEAL works in the areas of software visualization, evolution, and analytics. He co-authored more than 200 peer-reviewed articles and the book "Object-Oriented Metrics in Practice".



Rocco Oliveto is Associate Professor at University of Molise (Italy), where he is also the Chair of the Computer Science Bachelor and Master programs and the Director of the Software and Knowledge Engineering Lab (STAKE Lab). He co-authored about 150 papers on topics related to software traceability, software maintenance and evolution, search-based software engineering, and empirical software engineering. His activities span various international software engineering research communities. He has served as organizing and program committee member of several international conferences in the field of software engineering. He is also co-founder and CEO of *datasound*, a spin-off of the University of Molise aiming at efficiently exploiting the priceless heritage that can be extracted from big data. More information available at: https://dibt.unimol.it/staff/oliveto/.

Affiliations

Simone Scalabrino[1] · Gabriele Bavota[2] · Mario Linares-Vásquez[3] · Valentina Piantadosi[1] · Michele Lanza[2] · Rocco Oliveto[1]

Gabriele Bavota
gabriele.bavota@usi.ch

Mario Linares-Vásquez
m.linaresv@uniandes.edu.co

Valentina Piantadosi
valentina.piantadosi@unimol.it

Michele Lanza
michele.lanza@usi.ch

Rocco Oliveto
rocco.oliveto@unimol.it

1 University of Molise, Pesche IS, Italy

2 REVEAL @ Software Institute - Università della Svizzera Italiana, Lugano, Switzerland

3 Universidad de los Andes, Bogotá, Colombia