2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Data-Driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study

Simone Scalabrino*, Gabriele Bavota†, Mario Linares-Vásquez‡, Michele Lanza†, and Rocco Oliveto*
*University of Molise, Italy — ‡Universidad de los Andes, Colombia
†Università della Svizzera italiana (USI), Switzerland

***Abstract*—Android apps are inextricably linked to the official Android APIs. Such a strong form of dependency implies that changes introduced in new versions of the Android APIs can severely impact the apps' code, for example because of deprecated or removed APIs. In reaction to those changes, mobile app developers are expected to adapt their code and avoid compatibility issues. To support developers, approaches have been proposed to automatically identify API compatibility issues in Android apps. The state-of-the-art approach, named CID, is a data-driven solution learning how to detect those issues by analyzing the changes in the history of Android APIs ("*API side*" learning). While it can successfully identify compatibility issues, it cannot recommend coding solutions.**

We devised an alternative data-driven approach, named ACRYL. ACRYL learns from changes implemented in other apps in response to API changes ("*client side*" learning). This allows not only to detect compatibility issues, but also to suggest a fix. When empirically comparing the two tools, we found that there is no clear winner, since the two approaches are highly complementary, in that they identify almost disjointed sets of API compatibility issues. Our results point to the future possibility of combining the two approaches, trying to learn detection/fixing rules on both the API and the client side.

***Index Terms*—Android, API Compatibility Issues, Empirical Study**

I. Introduction

Android APIs are well know to be change-prone [1]–[4] because of the rapid evolution of the Android platform, which continuously provides users and developers with novel features. As a consequence of this fast evolution and of the high-dependability of mobile apps on the APIs [5]–[8], developers have to quickly react to newly released APIs to avoid issues related to API breaking changes.

In addition to breaking changes, fragmentation is also a well-known problem in Android [9]–[14]: 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 [9], [10], [15].

Both Android API evolution and fragmentation lead to *API incompatibility issues* in Android apps. Therefore, developers have to constantly watch for API release notes and identify how new and deprecated APIs should be used to avoid introducing bugs in the apps when running on specific environments. To illustrate an API incompatibility issue, let us introduce the Github issue 5059 of the `libgdx` framework (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 are common and have inspired approaches to automatically detect them [11], [16]–[18].

Those approaches detect API compatibility issues by relying on detection rules that are either hand-crafted or automatically mined from the API documentation. This latter is the solution adopted by CID [18], 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). 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.

In addition to presenting ACRYL, we investigated the benefits and weaknesses of "API side" and "client side" data-driven approaches, by comparing ACRYL with CID. The results after analyzing 11,863 snapshots of open source Android apps show that there is high complementarity between the two techniques, which points to the possibility of combining the two learning approaches in the future.

II. Background & Related Work

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

A. 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`. By defining those attributes,

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00055

developers indicate to the Google Play store which devices can download and install an app [19][1].

Each version of the Android API can include changes impacting, more or less severely, the apps' code. This includes 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 below:

```
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 }
```

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`) [20]. 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.

B. Related Work

Much research has been done on API misuses (*e.g.,* [21], [22]) and deprecated APIs [23]–[27]. 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 [2]–[4].

Besides the change- and fault-proneness of APIs, the problem of inaccessible Android APIs has also been recently studied by Li *et al.* [28]. 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 provided by 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.* [16] 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 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.* [17] 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 showed that 9k+ out of 10k analyzed apps suffer from misuses with outdated/abnormal APIs.

Device compatibility issues, and forward/backward compatibility issues are also 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.* [11] 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.

For this reason, Li *et al.* [18] 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.* [18], it ensures superior performance as compared to FICFINDER [11]. He *et al.* [29] 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*.

C. 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[2]. 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

[1] Note that 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.

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

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.

III. 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 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.

A. 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 $\texttt{API}_\texttt{i}$, otherwise use $\texttt{API}_\texttt{j}$). Both these tasks are not trivial and pose many challenges.

As explained in Section II, 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.

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

Fig. 1: Example of method to check the SDK version.

Figure 1 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 only 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. For example, if the `else` branch contains an invocation to method $\texttt{M}_\texttt{i}$ that invokes method $\texttt{M}_\texttt{j}$, and this latter invokes the Android API $\texttt{A}_\texttt{i}$, we must be able to link the usage of $\texttt{A}_\texttt{i}$ 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 [30]. Then, we use the WALA [31] 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 1 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 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 inter-procedural analysis of the conditional statement branches (*e.g.,* if/else branches) identifying all the calls to Android APIs[3] 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 III-C).

We do not consider CAUs having 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.

B. 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.

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 to 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, $A_t = A_f$ 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 III-C.

C. 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 represents 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.

[3] 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.

TABLE I: Types of API compatibility issues and suboptimal usages detected by ACRYL. 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.

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.

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 (≤ 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 2 shows an example of ρ. The black boxes contain 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 (*i.e.,* 11) — see the orange boxes in Figure 2. 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 V.

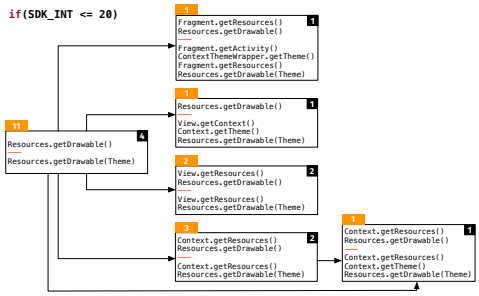

Fig. 2: Directed Acyclic Graph mapping the $\prec$ relationship.

D. 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 II), allows ACRYL to detect the types of API issues and suboptimal usages described in Table I. Table I 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=$\texttt{API}_2$.

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`.

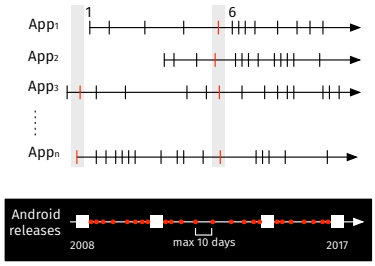

Fig. 3: Diagram used to explain the study design.

IV. Study Design

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

The verifiability of our study is guaranteed through a publicly available replication package [32] including the data used in the study as well as the ACRYL tool.

A. Research Questions

The study addresses the research question *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.

B. Context Selection

The first step to answer our research question is the selection of the subject mobile apps. We mined F-Droid [33], a catalogue of free and open-source Android apps, to identify all the 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.

C. Data Collection and Analysis

We adopt the study design depicted in Figure 3.

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 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 Figure 3. 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 Figure 3 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 $dates$=$\{d_1, d_2, \ldots, d_k\}$ 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 Figure 3, 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 done 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. The first buildable snapshot for the analyzed apps is from 2014/02/18. We release the list of apps/snapshots we considered [32].

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:

$$fixed_{\text{ACRYL}\cap CiD} = \frac{|fixed_{\text{ACRYL}} \cap fixed_{CiD}|}{|fixed_{\text{ACRYL}} \cup fixed_{CiD}|}\%$$

$$fixed_{\text{ACRYL}\setminus CiD} = \frac{|fixed_{\text{ACRYL}} \setminus fixed_{CiD}|}{|fixed_{\text{ACRYL}} \cup fixed_{CiD}|}\%$$

$$fixed_{CiD\setminus\text{ACRYL}} = \frac{|fixed_{CiD} \setminus fixed_{\text{ACRYL}}|}{|fixed_{\text{ACRYL}} \cup fixed_{CiD}|}\%$$

where $fixed_{\text{ACRYL}}$ and $fixed_{CiD}$ represent the sets of compatibility issues fixed by developers and detected by ACRYL and CID, respectively. $fixed_{\text{ACRYL}\cap CiD}$ measures the overlap between the set of fixed issues detected by both techniques, and $fixed_{\text{ACRYL}\setminus CiD}$ ($fixed_{CiD\setminus\text{ACRYL}}$) 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.

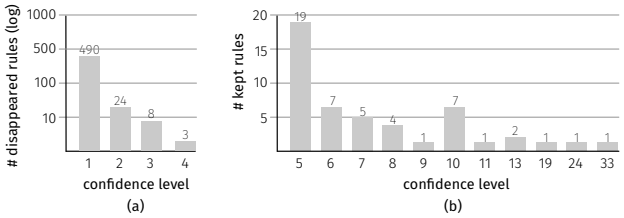

Fig. 4: Results of the tuning.

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.

1) 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.

V. 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 69 apps that could not be analyzed with CID from the comparison.

A. Tuning of the Confidence Level

Figure 4 (a) 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 $((\leq 10, \{\}, \{\texttt{Window.setFlags}\}), S)$ 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 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 Figure 4 (b) the distribution by confidence level of the rules that did not disappear detected in the tuning time period.

B. Comparison between ACRYL and CID

We report in Table II the comparison between ACRYL and CID, also showing the percentage of fixes for different categories of warnings. ACRYL achieves a slightly higher precision. Analyzing the results by category of warning, CID achieves a higher precision for Forward warnings compared to ACRYL, while the opposite happens for Backward warnings. It is worth noting that there is a single app (Ultrasonic) for which ACRYL reports many warnings (2,614, 834 of which are fixed). These warnings were fixed by updating the `minSDK` of the app. Excluding such an outlier, the overall precision of ACRYL drops to 7.0% (15.1% for Forward warnings and 5.9% for Backward warnings).

We report in Table III the detailed results for all the categories of warnings that ACRYL can detect. We do this both for (i) the apps used for the comparison and (ii) all the apps that ACRYL could analyze. Most of the warnings found by ACRYL belong to the macro-category "Backward". As previously mentioned, most of such warnings (specifically, "Backward Bug" warnings) come from a single app. However, even excluding this outlier, the warnings from the category "Backward Bug" are among the most frequently fixed ones (28%), 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 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 Figure 5 the distribution of the fixing time (in days) for both the approaches (we excluded the previously mentioned outlier for ACRYL). This 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 IV shows the overlap metrics of the fixed compatibility issues detected by the tools. Table IV clearly highlights that the two techniques are highly complementary. Indeed, 54.2% of the fixed issues are only detected by CID, 45.3% by ACRYL, and only 0.5% are in common.

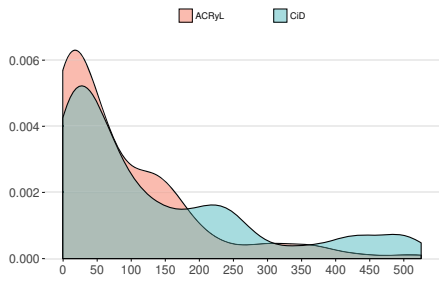

Fig. 5: Distribution of days needed to fix warnings.

C. Qualitative analysis

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 warning became a critical bug, since the API they used was not supported by Android versions before 21. This was quickly fixed on 2017/05/01 [34]. 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, instead, 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 [35]. 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 [36]. 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.

D. Discussion

Both *API-side*-learning (represented by CID) and *client-side*-learning (represented by ACRYL) approaches have their own advantages and disadvantages. 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.

TABLE II: Comparison between ACRYL and CID

	Backward			Forward			Total		
	#Fixed	#Warnings	Fix Rate	#Fixed	#Warnings	Fix Rate	#Fixed	#Warnings	Fix Rate
ACRYL	905	3,816	23.7%	25	166	15.1%	930	3,982	23.4%
CID	898	4,704	19.1%	201	1,222	16.4%	1,099	5,926	19.0%

TABLE III: Performance of ACRYL

		$\text{Apps}_{CiD} \cap \text{Apps}_{\text{ACRYL}}$			$\text{Apps}_{\text{ACRYL}}$		
		#Fixed	#Warnings	Fix Rate	#Fixed	#Warnings	Fix Rate
Backward	Bug	848	2,664	32%	848	2,681	32%
	Improvement	57	1,152	5%	59	1,220	5%
	Total	**905**	**3,816**	**23.7%**	**907**	**3,901**	**23.2%**
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		848	2,664	32%	848	2,681	32%
Non-severe warnings		82	1,318	6%	86	1,421	6%

TABLE IV: Overlap between ACRYL and CID

	Only by CID	Only by ACRYL	By CID and ACRYL
Backward	887 (49.6%)	890 (49.8%)	11 (0.6%)
Forward	201 (91.4%)	19 (8.6%)	0 (0.0%)
Total	1,088 (54.2%)	909 (45.3%)	11 (0.5%)

Client-side approaches offer several 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 qualitative analysis), 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 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.

VI. Threats to Validity

Construct validity: Identification of fixed API compatibility issues. 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 IV). For example, we do not consider an issue as fixed if the method affecting it was deleted in a subsequent snapshot.

Construct validity: Comparison with CID. To make the comparison between the two tools 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.

Internal validity: Calibration of the ACRYL*'s parameter.* We performed the calibration on snapshots belonging to a time interval not used in our study. The time needed to fix an issue reported in Figure 5 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.

External validity: Generalizability. Our study 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.

VII. Conclusions

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. For example, while *client-side* approaches allow to suggest fixing strategies, *API-side* approaches are easier to keep up-to-date.

Future work should focus on the definition of a hybrid approach combining both techniques.

REFERENCES

[1] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "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*, ser. ESEC/FSE. ACM, 2013, pp. 477–487.

[2] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM. IEEE Computer Society, 2013, pp. 70–79.

[3] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do API changes trigger stack overflow discussions? a study on the Android SDK," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC. ACM, 2014, pp. 83–94.

[4] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of API change- and fault-proneness on the user ratings of Android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2015.

[5] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the Android market," in *Proceedings of the 20th IEEE International Conference on Program Comprehension*, ser. ICPC, 2012, pp. 113–122.

[6] R. Minelli and M. Lanza, "Software analytics for mobile applications–insights & lessons learned," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR, 2013, pp. 144–153.

[7] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting Android reuse studies in the context of code obfuscation and library usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR. ACM, 2014, pp. 242–251.

[8] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE Software*, vol. 31, no. 2, pp. 78–86, 2014.

[9] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Proceedings of the 19th Working Conference on Reverse Engineering*, ser. WCRE, 2012, pp. 83–92.

[10] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in Android device driver customizations," in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. SP. IEEE Computer Society, 2014, pp. 409–423.

[11] L. Wei, Y. Liu, and S. C. Cheung, "Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2016, pp. 226–237.

[12] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM, 2013, pp. 15–24.

[13] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE. IEEE Computer Society, 2015, pp. 429–440.

[14] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "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*, ser. ICSME, 2017, pp. 399–410.

[15] P. Mutchler, Y. Safaei, A. Doupé, and J. Mitchell, "Target fragmentation in Android apps," in *Proceedings of the IEEE Security and Privacy Workshops*, ser. SPW, 2016, pp. 204–213.

[16] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, "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, 2017, pp. 678–690.

[17] T. Luo, J. Wu, M. Yang, S. Zhao, Y. Wu, and Y. Wang, "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*, ser. AIMS. Springer International Publishing, 2018, pp. 123–140.

[18] H. W. L. Li, Tegawendé F. Bissyandé and J. Klein, "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*, ser. ISSTA, 2018, pp. 153–163.

[19] Google. <uses-sdk>. https://developer.android.com/guide/topics/manifest/uses-sdk-element.

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

[21] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A benchmark for API-misuse detectors," in *Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories*, ser. MSR, 2016, pp. 464–467.

[22] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of Static API-Misuse Detectors," *IEEE Transactions on Software Engineering*, 2018.

[23] R. Robbes, M. Lungu, and D. Röthlisberger, "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*, ser. FSE. ACM, 2012, pp. 56:1–56:11.

[24] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "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*, ser. SANER, vol. 1, 2016, pp. 360–369.

[25] A. A. Sawant, R. Robbes, and A. Bacchelli, "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*, ser. ICSME, 2016, pp. 400–410.

[26] J. Zhou and R. J. Walker, "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*, ser. FSE. ACM, 2016, pp. 266–277.

[27] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated Android APIs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR, 2018, pp. 254–264.

[28] L. Li, T. F. Bissyandé, Y. L. Traon, and J. Klein, "Accessing inaccessible Android APIs: An empirical study," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME, 2016, pp. 411–422.

[29] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in Android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE. ACM, 2018, pp. 167–177.

[30] Dex2jar. http://code.google.com/p/dex2jar.

[31] Wala. http://wala.sourceforge.net/.

[32] S. Scalabrino, G. Bavota, M. Linares-Vásquez, M. Lanza, and R. Oliveto. Replication package. https://dibt.unimol.it/report/acryl-msr/.

[33] F-droid. https://www.f-droid.org/.

[34] Kandroid, commit 0b0d0. https://git.io/fh8uX.

[35] Dandelion, commit af007. https://git.io/fh8uP.

[36] Rick app, commit 05efd. https://git.io/fh8u6.