CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps

Li Li
FIT, Monash University
Australia

Tegawendé F. Bissyandé
SnT, University of Luxembourg
Luxembourg

Haoyu Wang
Beijing University of Posts and Telecommunications
China

Jacques Klein
SnT, University of Luxembourg
Luxembourg

ABSTRACT

The Android Application Programming Interface provides the necessary building blocks for app developers to harness the functionalities of the Android devices, including for interacting with services and accessing hardware. This API thus evolves rapidly to meet new requirements for security, performance and advanced features, creating a race for developers to update apps. Unfortunately, given the extent of the API and the lack of automated alerts on important changes, Android apps are suffered from API-related compatibility issues. These issues can manifest themselves as runtime crashes creating a poor user experience. We propose in this paper an automated approach named CiD for systematically modelling the lifecycle of the Android APIs and analysing app bytecode to flag usages that can lead to potential compatibility issues. We demonstrate the usefulness of CiD by helping developers repair their apps, and we validate that our tool outperforms the state-of-the-art on benchmark apps that take into account several challenges for automatic detection.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**;

KEYWORDS

Android, Framework Base, API-related Compatibility Issue, CiD

ACM Reference Format:
Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213857

1 INTRODUCTION

Modern software systems are typically designed following a model in which the core of the system (e.g., SDK) is usually accessible via an Application Programming Interface (API) that lists the set of "entry points" to the system. While this model allows to easily leverage the core of the system, any evolution of the API can lead to compatibility issues which may inconvenience end-users. Android is such a system where apps rely on framework APIs to access Android stack functionalities on the device, ranging from inter-application communication facilities [1, 2] to hardware interactions. The Android framework, however, evolves rapidly, with about 300 releases shipped in a few years on tens of thousands of device models [3]. Thus, although Android implements a mechanism for enabling app developers to specify the API level on which their apps should run, it is common to see reports of compatibility issues faced by users when running apps: runtime crashes with error messages often revealing that there is no such API methods or the behaviour of such methods is not initially expected by developers [4–6].

In the Android ecosystem, developers have limited control on the device where their apps will be run. With the *minSdkVersion* and *maxSdkVersion* manifest parameters, developers can constrain the Android versions, hence the API levels, that their apps are going to support. Unfortunately, the practice of setting such attributes is not yet clearly established in the development community. For example, We have found that the *minSdkVersion* attribute is not taken into account by 9 F-Droid apps [7] despite a strong recommendation in Android documentation to app developers for setting this attribute in all apps. Indeed, not applying the minimum API level or applying with a wrong level (e.g., lower than the least needed level) can lead to scenarios where backward compatibility issues will arise: some APIs used in the app code are actually not available in some older versions of Android, resulting in unfortunate crashes. When the *minSdkVersion* attribute is not specified in an app, any Android operating system would accept its installation since no runtime compatibility issues can be foreseen at that time. In contrast, the Android documentation does not recommend to set the *maxSdkVersion* attribute. Yet, not setting the maximum API level may still lead to latent forward compatibility issues as some used API methods may be deprecated in newest versions of the platform. Actually, the recommendation is due to the fact that, in order to support forward-compatibility, Android maintainers simply hide (e.g., with *@hide* annotation, making the method unavailable during compilation, but reachable in the on-device stack) the API method instead of completely removing it from the framework. As a result, it is no longer accessible from the developer's point of view (because the API is no longer in the public SDK) but still accessible at runtime (because it is still available in the framework[1]). Unfortunately, as demonstrated by Li et al. [8], hidden Android APIs are also subject to removal or invasive changes due to the rapid evolution of Android systems. Thus, the recommendation to Android developers on forward compatibility may actually lead to more latent issues when deprecated APIs no longer meet their usage contract.

From a market maintainer's point of view, a proliferation of apps with API compatibility issues can be detrimental in securing the loyalty of a too-often inconvenienced user base. This is especially harmful when the market is under competition with others (e.g., in regions such as China where the official market is not predominant). Thus, it could be a key asset for market maintainers to systematically analyze and identify app compatibility issues that could be sent automatically to developers to ease and enforce the fixes.

A few works in the literature have investigated API compatibility issues [9–12]. Most recently, Wei et al. have focused on fragmentation-induced compatibility issues [9]. They have empirically analyzed some issue reports to **manually** build a model expressing common patterns of compatibility issues related to Android fragmentation. The excerpt below shows the built model for an example API: besides the signature of the flagged API method, the module provides additional information in the issue report where a relevant compatibility issue was filed.

```
"APISignature":"<android.view.View: void setAlpha(float)>",
"conditions": { "SDK":"11" },
"additionalInfo": "The API is introduced in API 11, [URL]"
```

Unfortunately, as explicitly acknowledged by the authors, their FicFinder approach implies a labour-intensive process with practical limitations for manually identifying, extracting, and building models for a large number of issue reports. In the current prototype, the authors have managed to model only 20 API methods that may cause compatibility issues: this coverage is very minimal given that thousands of methods are relevant to the API compatibility problem.

Our work, therefore, generalizes FicFinder to all API compatibility issues, aiming at being **systematic** in the mining of Android framework versions to model the lifecycle of all API methods, and being more thorough in the analysis of apps to detect potentially problematic usages with **reduced false positives**. We expect our approach to be used to (1) provide better user experience. When a user downloads/installs an app, our approach can be applied beforehand to check if the app is compatible with her/his device. Otherwise, users would have wasted time and bandwidth to install an app that cannot be properly used. (2) support market maintainers. With our approach, market maintainers can have an overview on the API ranges that their hosted apps support and thus can recommend necessary updates that developers must perform for their apps or even purge some problematic apps (e.g., incompatible to majority devices) for avoiding bad user experiences. (3) identify potential developer mistakes. As experimentally illustrated in the evaluation section, app developers are likely to make mistakes on compatibility issues, especially when @TargetApi, @SuppressLint("NewApi") annotations are incorrectly used (i.e., modern IDEs are not capable of highlighting such compatibility issues).

[1]The public SDK and the framework code running on a device are different although they are compiled from the same source code.

Overall, this paper reports on the following contributions:

- We provide an overview of Android API evolution to quantify the extent of cases when compatibility issues may arise in the Android ecosystem. We also showcase real-world examples of API-related compatibility issues to qualify their impact during app execution.
- We design and implement CiD, an approach for identifying compatibility issues directly from Android app bytecode (because generally source code is difficult to obtain) based on a thorough modelling of API lifecycle and a static analysis of APIs within app code.
- We demonstrate the efficiency of CiD by i) detecting compatibility issues in real-world apps, ii) outperforming state-of-the-art tools on benchmark apps and iii) producing issue reports that are acknowledged and fixed by development teams. Developers of seven open-source Android apps have confirmed the compatibility issues that we submitted to them (through issue reports). The fix was immediately performed for two of the apps.

2 ANDROID API COMPATIBILITY ISSUES

2.1 Background on API Levels

Android implements a scheme based on API levels to manage app compatibility across the variety of devices operated by a large range of OS versions. As the Android operating system evolves, each released version is associated with an API level representing a unique identifier for a set of functionalities that are implemented. Actually, the practice in the Android ecosystem is that each version is referred to in several ways: its Android version number (e.g., Android 2.3.2), a code name (e.g., Nougat), or the associated API level (e.g., API level 24) [13].

When programming apps using the Android Software Development Kit (SDK), developers leverage API routines available in the provided frameworks which are each associated with an Android version, and thus targets each a specific API level. Although an app may be developed and tested against a specific framework, developers can leverage the API level identifier to indicate which version of the platform they require [14]. Similarly, API level is used to negotiate the installation of apps on user's devices to mitigate compatibility issues. Typically, an app Manifest includes three attributes for these specifications:

- *minSdkVersion*, i.e., the minimum API level on which the app is designed to run. Google Play leverages this attribute to filter out such devices that have a lower SDK platform version than the declared value of minSdkVersion. In other words, users, who have a device with a platform version lower than minSdkVersion, will not even see this app on Google Play.
- *targetSdkVersion:* The most appropriate API level on which the app is designed to run. If this attribute is not explicitly set, the default value will be set to the value of minSdkVersion.
- *maxSdkVersion:* The maximum API level on which the app is designed to run. As explicitly indicated in documentation [15], in order to be forward-compatible (or backward-compatible from the Android framework point of view), declaring this attribute is not recommended by Google.

2.2 Example Issues

Despite regular changes in its API, Android development team is taking a few steps to ensure that, in general, forward and backward compatibility issues can be avoided. Consider the illustration in Figure 1.

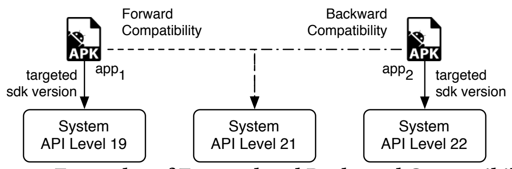

Figure 1: Examples of Forward and Backward Compatibility.

- *Forward Compatibility* implies that a given app developed with API level 19 as the target could be executed seamlessly on devices running Android systems at API level 21 and potentially above. It is officially advertised [16] to app developers that forward compatibility is assured since almost all changes to the API are additive. Android documentation has even recommended developers to not set the maxSdkVersion.
- *Backward Compatibility* implies that a given app developed with API level 22 as the target should be able to execute normally on devices running Android systems at API level 21 and potentially below. The Android developer documentation explicitly warns that Android apps are not necessarily backward compatible [17].

2.2.1 *Forward compatibility issues.*

On one hand, forward compatibility is important in the Android ecosystem since most Android devices (notably smartphones) receive over-the-air updates, and previously installed apps may fail to run properly when the system API level changes. Although Android documentation emphasizes that changes to the API do not generally threaten forward compatibility, we have come across several cases where API changes for improving security or API robustness can create compatibility issues. In the example of Listing 1, the Wikipedia app, listed in the F-Droid repository, uses an API method which was removed since API level 19. This app, however, is still supposed to run on most recent versions of Android. This is possible as the concerned API method is actually only removed from the public SDK. In reality, the method remains available in the software stacks (i.e., the framework) shipped with devices.

```java
public abstract class SwipeableBottomDialog extends
     DialogFragment {
public void setContentPeekHeight(int height) {
 contentPeekHeight = height;
 if (listView != null) {
  //This API is removed after API level 19
  listView.setSelectionFromTop(SPACE_VIEW_POS,
     -contentPeekHeight);
}}}
```

Listing 1: An Examples of Forward Compatibility Issue. The Code Snippet is Extracted from a Real App named Wikipedia, which is currently available in F-Droid.

This guarantee of availability of API methods in new versions may however involve latent issues as there might be legitimate reasons to remove an API. For example, as described in Listing 2, the behaviour of an API may change in new releases, creating forward compatibility issues. In this case, app code must be rewritten to handle the newly thrown exception.

```java
//(1) original definition
public final int getAssetInt() {
 return mAsset;
}
//(2) removed from the public SDK
ci: f8f09a15a409f373f22aa475bb0defd264088e4f
Hide AssetInputStream.getAssetInt
//(3) added it back again to the public SDK
ci: b1bd1fe7fd9ed6b6e4518713ef5f5716a84d97e8
Revert "Hide AssetInputStream.getAssetInt."
//(4) change the API's behavior
ci: 2d19d202bdf6c16b8e01b73f3a742b2670bff907
Make getAssetInt throw unconditionally.
public final int getAssetInt() {
 throw new UnsupportedOperationException();
}
```

Listing 2: Evolution samples of API *getAssetInt()* of class *AssetInputStream.*

We now enumerate cases where API changes actually involve removals or invasive changes which, although they may not lead to method availability errors, will maintain poor user experience in terms of security, robustness or even unexpected behavior:

- In API level 23, Apache HTTP Client class was entirely replaced by the HttpURLConnection class to improve performance, since the latter reduces network usage (e.g., through transparent compression and response caching) and minimizes energy consumption.
- To provide users with greater data protection, starting from API level 23, Android removes programmatic access to the device's local hardware identifier for apps using the Wi-Fi and Bluetooth APIs: in recent versions, API methods *WifiInfo.getMacAddress()* and *BluetoothAdapter.getAddress()* now return the default constant value of 02:00:00:00:00:00. This behavior change has triggered a lot of stack overflow discussions [18].
- To facilitate multitasking, API level 21 implements new features for concurrent documents and activities. In this context, the API method *ActivityManager.getRecentTasks()* became problematic as it could leak personal information about documents. Thus, this API method had to be deprecated in public SDK, and modified in on-device stack to return a small subset of data.

Even if Google claims that Android apps do not suffer from forward compatibility issues, mainly because removed APIs are still kept in the framework side as hidden APIs, forward compatibility induced APIs are still encouraged to be replaced because of security and performance concerns. Furthermore, since hidden APIs are also subject to remove or change, forward compatibility is also not fully guaranteed in practice anyway.

2.2.2 *Backward compatibility issues.*

On the other hand, with the rapid evolution of Android frameworks, backward compatibility is often not met for new generation of apps on old devices. Generally, in such scenarios, apps crash when the old system fails to locate the called method. Listing 3 describes the case of a real-world app, *YASFA1* developed as a simple forms app [19], which uses an API method only available starting at API level 11. On any older systems, which however are marked as compatible with the app (by setting the *minSdkVersion* attribute), execution of this simple app will terminate with a crash (i.e., lines 11-14).

```
 1 public class FButton extends Button {
 2 public boolean onTouchEvent(MotionEvent event) {
 3  //setAlpha(float) is introduced at API level 11
 4  if (event.getAction() == MotionEvent.ACTION_DOWN) {
 5   ... ...
 6   setAlpha(0.4f);
 7  }else if (event.getAction() == MotionEvent.ACTION_UP) {
 8   setAlpha(1);
 9 }}}
10 //Crash with No Such Method Error
11 java.lang.NoSuchMethodError: com.yasfa.views.FButton.setAlpha
12  at com.yasfa.views.FButton.onTouchEvent(FButton.java:42)
13  ... ...
14  at dalvik.system.NativeStart.main(Native Method)
```

Listing 3: An Example of Backward Compatibility Issue. The Code Snippet is Extracted from a Real App named YASFA1, which is currently available in F-Droid. API *setAlpha* is defined in class *View* from whom class *Button* (or *FButton*) extends.

As recommended by Google and shown in Listing 4, in order to keep backward compatibility, developers should check at runtime the supported API level of the running device (e.g., line 1) and then decide whether or not to access the backward compatibility induced APIs. Actually, as what we will show in our evaluation section, even if developers know that they need to protect compatibility-induced APIs, it is still not trivial for inexperienced developers to take care of all the accessed problematic APIs, resulting in still compatibility issues.

```
 1 //App Paper-Wallet (Available in F-Droid)
 2 //Example (1)
 3 if (Build.VERSION.SDK_INT >= 23) {
 4  //getColor() is introduced at level 23
 5  return context.getColor(id);
 6 } else {
 7  return context.getResources().getColor(id);
 8 }
 9 //Example (2)
10 if (Build.VERSION.SDK_INT >= 11 && clipboardListener != null) {
11  clipboardHelper. removeClipboardListener(clipboardListener);
12 }
13 public void removeClipboardListener(Runnable runnable) {
14  if (runnable != null && listeners.containsKey(runnable)) {
15   //removePrimaryClipChangedListener() is introduced at level
       11
16   clipboard.removePrimaryClipChangedListener(
       listeners.get(runnable));
17 }}
18 //Example (3)
19 AlarmManager am = null;
20 if (Build.VERSION.SDK_INT >= 19) {
21  am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
22 }
23 if (null != am) {
24   //isEncrypted() is introduced at API level 19
25    am.setExact(-1, -1, null);
26 }
```

Listing 4: Examples of Protecting the Access of Problematic Android APIs.

Accessing backward compatibility induced APIs, without proper protection, will simply result in app crashes, giving poor experience to end-users.

2.3 Evolution of Android APIs

We further explicit the appearance of API compatibility issues by studying the evolution of Android APIs.

2.3.1 Dataset Collection. To harvest all API versions we consider the source code of the Android framework[2]. We extract all public API methods, including the publicly accessible constructor methods. At the moment, there are over 300 releases[3] recorded on the Android code base repository. Since, as previously discussed, several releases can be associated to the same API level when the accompanying changes (e.g., critical bug fixes) do not significantly impact the API set, we focus on a single release per API level. In total, we have considered 24 releases for covering API levels 1 through 25 (level 20 remains for wearable devices only). We then build the mapping between each API level and its associated public APIs.

2.3.2 Data Analysis. First, we compute the additions and removals of APIs between consecutive API levels, from which we observe that most changes to the public API are additive for introducing new functionality. Nevertheless, we note that in some evolution (e.g., from API level 22 to API level 23), up to 6% (1921/30540) of the API set has been removed.

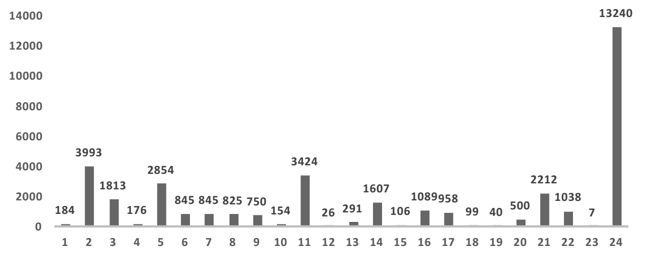

Figure 2: Life expectancy of public Android APIs. X-axis corresponds to the number of API level generations before an API method is removed from the code base. Y-axis presents the number of APIs removed.

We then look into the statistics of changes beyond consecutive releases by studying the life expectancy of public APIs within the framework. An API method age is computed as the number of API levels where the method is publicly accessible. Figure 2 shows that most APIs indeed have always been available in the framework since the beginning. However, a significant proportion of API methods stay available only at a few API levels. The use of such API methods can thus lead to compatibility issues, either because they are not yet available in the on-device Android stack, or because their behaviour, which is either insecure or non robust, has warranted their removal from the public API.

Finally, our investigation into the evolution of APIs has also highlighted the case of several APIs that are removed from and then re-introduced again into the Android framework code. Such modification scenarios can cause latent compatibility issues as they are often accompanied with significant changes in the API behaviour. Table 1 describes examples of such APIs.

[2]Android framework code is open-sourced and is available at [3].
[3]A release is marked with a Git Tag on the Github repository.

Table 1: Android APIs that are Added Back after Removed in an Old Release.

API	Introduced	Removed	Added Back
Gravity.getAbsoluteGravity	14	16	17
KeyEvent.getDeviceId	1	9	12
MotionEvent.getDeviceId	1	9	12
DatagramSocketImpl.getOption	1	5	6
DatagramSocketImpl.setOption	1	5	6
SocketImpl.getOption	1	9	14
SocketImpl.setOption	1	9	14

3 APPROACH

In view of the illustrative example issues provided above, to avoid API compatibility issues, one should have sufficient knowledge on the lifecycle of a given API. Then, after analysing the API call site and the associated conditions for the call to occur, it is possible to warn on potential issues. We contribute with a Compatibility Issue Detector (CiD) approach, which not only aims at flagging potential API compatibility issues but also at helping developers understand better the identified compatibility issues. CiD performs static analyses on both app and framework code to mine compatibility-related issues.

Figure 3 depicts the three modules of CiD. The first module, ALM, builds the API lifecycle model based on a mining of Android framework revision history. In the second module, AUE, an analysis is performed to locate and extract the usage schema of Android APIs in an app. This module considers not only core app code but also any to-be dynamically loaded code available in the app package. Finally, the analysis keeps a summary of the conditions under which the extracted APIs could be reached. Finally, the third module, ACA, evaluates the output of ALM and AUE altogether to flag any potential API compatibility-related issue. We provide further details on the working process of these modules in the remainder of this section.

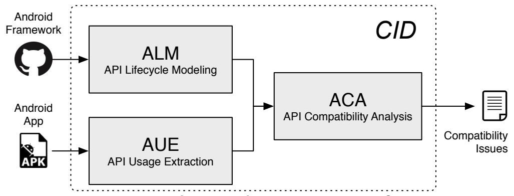

Figure 3: The working process of CiD.

3.1 API Lifecycle Modeling (ALM)

The goal of the ALM module is to produce a reliable and reusable model that can be queried by and be integrated into any other tool/approach where there is a need to query lifetime information of a given Android API method. Let us consider the case example of API method *isDefaultNetworkActive()* from class *android.net.ConnectivityManager* in the Android framework. By querying ALM, one would quickly be informed that this API method has been introduced with the release of API level 21 and is still present in the latest release of Android. Building a complete and reliable lifecycle of the API is however challenged by various programming facilities:

- **Inheritance.** The first analysis challenge for ALM is related to Java's inheritance. In the framework code, a sub-class can redefined API methods inherited from its super-class. Thus, when this redefined method is dropped out, ALM should keep track that the API may still be available for this class since it is still implicitly inherited from super-class with unchanged implementations.
- **Generic type.** Besides Java inheritance, which may complexify the computation of an API's lifetime, the existence of *Generic types* can also impact the accuracy of API lifecycle modelling. As an example, *set()* of class *LinkedList* (line 2 in Listing 5), is an API that contains a generic type (*E*). A simple syntactic matching between this collected signature with app code cannot readily allow to find cases of usage (e.g., lines 3,4 in Listing 5).
- **Varargs.** Similar to API methods with generic types, some API methods use varargs which will later challenge the detection of their usage in Android apps. For instance, API *remove(long...)* of class *DownloadManager* has a varargs parameter (line 6 in Listing 5). The real usage of this API could be *remove(long)* or *remove(long,long)* (lines 7,8 in Listing 5), and in any case will be syntactically different from the method signature.

```
1 //Generic Programming
2 <java.util.LinkedList: E set(int,E)>
3 <java.util.LinkedList: String set(int,String)>
4 <java.util.LinkedList: String set(int,Double)>
5 //Varargs
6 <android.app.DownloadManager: int remove(long...)>
7 <android.app.DownloadManager: int remove(long)>
8 <android.app.DownloadManager: int remove(long,long)>
```

Listing 5: Generic Type and Varargs Related APIs and Their Example Instances.

3.1.1 *ALM working process.*

Figure 4 illustrates the working process of ALM in three steps:

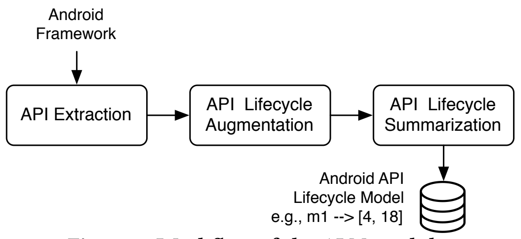
Android Framework
API Extraction
API Lifecycle Augmentation
API Lifecycle Summarization
Android API Lifecycle Model e.g., m1 --> [4, 18]

Figure 4: Workflow of the ALM module.

API extraction. We have collected all the public Android APIs (including the publicly accessible constructor methods) from the source code of the Android framework code base for 24 releases, covering API levels from 1 (Oct. 2008) to 25 (Dec. 2016). The API lifecycle modelling is then based on this history.

API information augmentation. To overcome the inaccuracies and misses that may unfold because of the challenges enumerated above (i.e., inheritance, generic type and varargs), ALM includes an analysis step, which augments the collected API signatures with information on the inheritance tree, the potential concrete parameters and their associated types.

API lifecycle summarization. Finally, the API lifecycle is summarized by differentiating every pair of API sets extracted for continuous API levels (e.g., API level x and $x + 1$). In this way, we can accurately identify which APIs are introduced or removed in version $x + 1$, and model the lifecycle of all Android APIs throughout all versions. The yielded model can then be queried to establish for example that the lifetime of API *hasTransientState()* of class *android.view.View* starts at API level 16 and ends at API level 25.

We remind the reader that the implementation of the ALM module as well as its accompanying query interface is independent of the implementation of CiD and is thus reusable for other approaches and tools. The current implementation of CiD has actually provided a query interface for facilitating such reuse. Last but not the least, the ALM module is implemented in a fully automated manner. Hence, it can be automatically and regularly updated following the updates of Android framework.

3.2 API Usage Extraction (AUE).

Given a third party developer app, CiD must first identify all API methods that are leveraged in its code. To that end, we implement the AUE module which extracts, from app bytecode, all the accessed Android APIs, including such API methods that are called within to-be dynamically loaded code. Figure 5 shows the working process of AUE which is carried out in three steps: i) Locating additional code (i.e., outside the main *classes.dex* code, ii) Building a conditional call-graph (i.e., taking into account runtime checks of API levels), and iii) Resolving API usage (i.e., collecting usage locations and contexts of API methods).

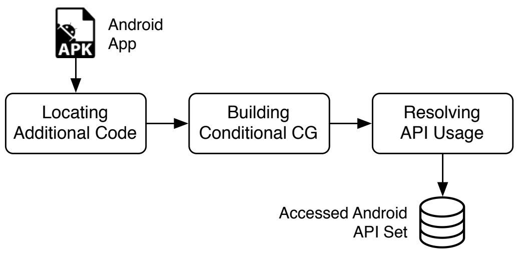

Figure 5: The working process of module AUE.

Locating Additional Code. In addition to the primary app code, which is assembled into *classes.dex* and located in the root directory of a given app packaged file (i.e., apk), an Android app may also include extra code that developers often hide in some separate archive files packaged with the rest of items in the APK file. Such additional code can then be loaded and executed at runtime via the so-called dynamic code loading (DCL) mechanism. Since additional code will also access Android APIs, a complete analysis should take it into consideration in order to avoid missing compatibility issues that may lead to runtime crashes.

With the AUE module, we focus on additional code that is statically available in APK file[4]. We reuse an existing heuristics-based approach by Li et al. [20] to locate to-be-dynamically loaded code: Given an app a, we unzip it and traverse all its embedded files, if a given traversed file is an archive file (the file extension could vary from dat, bin to db), we recursively look into it, to check if it contains a dex file through its magic number (035). All retrieved dex files (usually they also come as classes.dex) are then considered for extracting APIs.

[4]Some additional code may not be statically available as it is downloaded at runtime from remote servers.

Building a Conditional Call Graph (CCG). Detecting the call to an API method within app code does not necessarily warrant a check of this API lifetime to ensure that compatibility issues may not arise. Indeed, developers often take steps, directly in the code, to consolidate the access of actual calls on user's devices by checking dynamically the API levels as recommended in the Android development guidelines. In other words, even if a given Android app has accessed a problematic Android API, which may induce API compatibility issues, we cannot report it as such without checking if it is somehow protected by condition checkers.

The AUE module performs a backward **data-flow** analysis to verify whether an API method is called under API level-related conditions. In practice, the conditions for accessing an API may not be identified in a straightforward manner. Some conditions may be set well above in the call stack hierarchy or be set only for constructing objects. Therefore, the backward data-flow analysis must be **inter-procedural** and be aware of **constructor methods** in order to avoid false positive results. Since the version check is usually done with condition statements, the data-flow analysis also needs to be **path-sensitive**.

To simplify the analysis process by AUE, we construct a special call graph, which we refer to as a *conditional call graph* (CCG), to support the inter-procedural, path-sensitive, constructor-aware backward data-flow analysis. A CCG is defined as a tuple (V, E, C, f), where V is a set of methods representing vertexes of the graph, E is a set of directed edges connecting two methods (e.g., for edge $v_1 \rightarrow v_2$, v_1 is the caller while v_2 is the callee), C is a set of conditions related to API level, and $f : E \rightarrow 2^C$ is a function assigning each edge a subset of conditions[5]. Given an edge $e_1 : v_1 \rightarrow v_2$, $f(e_1) = \{c_1, c_2\}$ means that there are at least two different call paths from method v_1 to method v_2 (one with condition c_1 and the other with condition c_2). Figure 6 presents an example of CCG, which is built by statically analysing the snippet code shown in Listing 4.

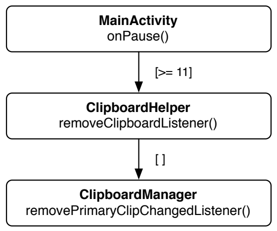

Figure 6: Simplified Conditional Call Graph built by CiD for the snippet code shown in Listing 4.

Resolving API Usage. Given the list of called API methods and information on conditions that are applied to their call paths, the AUE module can now accurately resolve the API usage. For additionally identified dex files, we follow the same process to visit all their statements in order to harvest all the referenced Android APIs. So far, since we mainly focus on publicly accessible APIs (e.g., available in the Android SDK), we have not taken into account reflective calls. For future work, we plan to integrate the reflection analysis module of DroidRA [20] into this work so as to be also aware of reflectively accessed APIs.

[5]2^C is the powerset of C, i.e., the set of all subsets of C (including the empty set and C itself)

3.3 API Compatibility Analysis (ACA).

Given an Android app, the output of ALM (based on analysis of the framework) and the output of AUE (based on analysis of the app bytecode), the ACA module matches lifecycle models with information on the target API levels.

Each resolved API reported by AUE is queried against the API lifecycle model interface which provides the lifetime of the API. Then, ACA compares the lifetime of accessed APIs to the platform constraints (cf., minSdkVersion) declared by developers in the app manifest, and also with regards to the conditions under which they are called as summarized by the conditional call graph built by the AUE module. Let us take the snippet shown in Listing 4 again as an example, although API *removePrimaryClipChangedListener()* is introduced at level 11 and the declared *minSdkVersion* is less than 11, thanks to the CCG, CiD knows that the API is accessed with protections and therefore will not report it as a potential compatibility issue. Based on this analysis, CiD can flag some API usages as problematic and thus report them to the app developers that their app may present some potential API-related compatibility issues.

4 EVALUATION

In this work, we are interested in answering the following research questions:

- **RQ1:** *Is CiD effective in API compatibility issue detection?* We investigate the warnings of CiD on real-world apps to report on potential false positives.
- **RQ2:** *Are compatibility issues addressed by developers during app updates?* We investigate the output of CiD for consecutive versions of apps to quantify and qualify how API compatibility issues have been fixed.
- **RQ3:** *Can CiD provide useful information for app developers to facilitate the diagnosis and repair of API compatibility issues?* We conduct a live study by filing bug reports to development teams based on the warnings yielded by CiD on open-source apps.
- **RQ4:** *How does CiD compare with existing tools?* Finally, we build a benchmark dataset of API-related compatibility issues in Android apps taking into account different challenges for detecting them, to evaluate the performance of CiD as well as the most recent state-of-the-art, FicFinder [9].

4.1 RQ1: Issue Detection Effectiveness

To evaluate the effectiveness of CiD in detecting API compatibility issues, we run CiD on a dataset of open-source apps collected from the F-Droid repository. We have considered for our experiments the latest versions of 1,797 apps available in the dataset[6]. Overall, CiD has reported a total of 21,002 forward and backward compatibility issues for 891 apps. We also recorded, thanks to the Conditional Call Graph built in the AUE module, 6,252 additional cases where developers carefully protected APIs with condition checks on the API level of the running device to avoid compatibility issues. This check is generally performed against the attribute *Build.VERSION.SDK_INT*. As marginal cases, we also identified 163 apps which perform the API level checks with the

[6]Representing all the available apps on Dec. 2016.

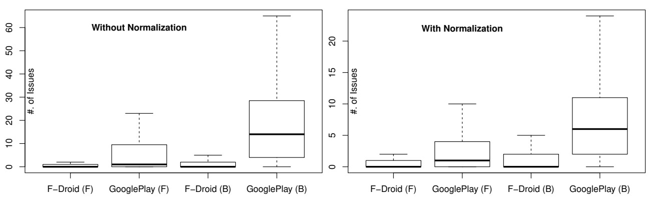

Figure 7: Distribution of the Number of Reported Compatibility Issues between F-Droid (open-sourced) and Google Play (close-sourced) Apps. (F) Stands for forward compatibility issue warnings. (B) Stands for Backward compatibility issue warnings

Build.VERSION.SDK attribute which is deprecated since API level 4. This is an interesting case where even the check condition code added by developers presents itself forward compatibility issues.

Among the 21,002 unprotected issues reported, 11,247 are forward compatibility warnings raised for 579 apps and 9,755 are backward compatibility issues raised for 609 apps. Given the number of warnings raised, and in the absence of third-party data or system to validate such warnings, we resort to verify some of the reported issues through executing the apps with manual interactions, e.g., observing crashes as in the following case studies:

Case Study: org.vi_server.red_screen. CiD reports a compatibility issue for this app after detecting that it has accessed the *setSystemUiVisibility()* API method without any runtime API level check. CiD further details that (1) this API method is only available from level 11 onwards and (2) it can be reached through the *onCreate* method of component *RedScreenActivity*. Based on this information, we are able to launch and explore the app for reaching the aforementioned API and to expose the problem with an execution crash. We then harvest the error messages through *adb logcat* command, which confirms that this crash is due to API compatibility issue:

```
No such method error for
```
android.view.View.setSystemUiVisibility()[7]

Comparison between open and close sourced apps. Since the AUE module of CiD works on bytecode of Android apps, we are also capable of checking for compatibility issues on closed-source apps from Google Play repositories. We focus on versions of apps compiled in 2016, and randomly select 1,797 apps to compare the distribution of API compatibility issues between open-source and close-source apps. Figure 7 presents the distributions for both datasets with and without normalizing based on app size. The median value for F-Droid (F), Google Play (F), F-Droid (B) and Google Play (B) are respectively 0, 1, 0, 14 issues per app, and 0, 1, 0, 6 issues per megabyte of app code. We have checked with the Mann-Whitney-Wilcoxon (MWW) test [21] that the difference of median values between F-Droid and Google Play is statistically significant for both forward and backward compatibility issues: Close-source apps have more compatibility issues than open-source apps.

RQ1: CiD is effective in detecting API compatibility issues. Identified backward compatibility issues deserve more attention from developers as they can lead to runtime crashes.

[7]Note that this app does not crash on devices running latest platform versions. This problem has been fixed immediately after we reported it to its developers.

4.2 RQ2: Compatibility Fixes by Developers

Another means of assessing the warnings yielded by CiD is to investigate the fixes performed by app developers in the history of app updates. Since most developers do not consistently mention in changelogs all fine-grained details on the changes performed, we focus on verifying how and why CiD warnings are different from one version to another. To that end, we leverage the AndroZoo [22] dataset and build app version lineages for randomly selecting 2000 cases of updates. Each case is represented as a pair of apps (app_1, app_2) sharing the same package name and signed with the same developer certificate. In a given app lineage, app_2 is the immediate successor of app_1: app_2 has a higher[8] version code than app_1.

We run CiD on both apps of all 2,000 pairs involved in this study. Because of exception raised during the static analysis (e.g., due to timeout or known bugs of Soot) performed in the AUE module for some apps, our investigation was eventually limited to 1,613 version pairs. Among these pairs, the difference between the outputs of CiD shows that 256 update cases have made changes that eliminate API compatibility warnings. More concretely, a difference occurs when, for a given API, CiD outputs a warning on app_1 but not on app_2.

Compatibility issues are fixed either by removing/replacing the API method or by inserting API level condition checks to prevent any execution on devices where issues can arise. Table 2 further summarizes the top five fixed APIs by removal/replacement or condition checking for both forward and backward compatibility issues. We observe that the top API methods whose usage are being protected with condition checks are also the same that are simply removed/replaced. We further note that a given API which is problematic (in the sense that its usage may lead to compatibility issues) is more likely to be removed/replaced than protected via condition checks. This finding is in line with the expectation of android maintainers: putting a condition check on the usage of an API method does not fully take into account the security and/or performance requirements that justify its deprecation.

Table 2: Top 5 Fixed APIs (via Removal/Replacement or Condition Check Insertion) in Real-word Android Apps.

Removal			Protection		
API	**Life**	**Counts**	**API**	**Life**	**Counts**
Forward Compatibility					
Notification.setLatestEventInfo	[1,22]	24	HttpEntityEnclosingRequestBase.setEntity	[1,22]	10
HttpEntityEnclosingRequestBase.setEntity	[1,22]	19	HttpResponse.getStatusLine	[1,22]	8
BasicClientCookie.setDomain	[1,22]	10	StatusLine.getStatusCode	[1,22]	8
HttpEntity.getContent	[1,8]	10	HttpEntity.getContent	[1,22]	8
Array.newInstance	[1,22]	10	Array.newInstance	[1,8]	7
Backward Compatibility					
View.setAlpha	[11,25]	20	View.setAlpha	[11,25]	12
ViewPropertyAnimator.setDuration	[12,25]	14	ViewPropertyAnimator.setDuration	[12,25]	11
ViewPropertyAnimator.alpha	[12,25]	14	ViewPropertyAnimator.alpha	[12,25]	11
View.animate	[12,25]	13	View.animate	[12,25]	10
Editor.apply	[9,25]	12	View.setLayerType	[11,25]	3

RQ2: During app updates, developers actually fix compatibility issues that CiD would have helped to flag. It should be noted that the practice is more to remove/replace API methods in app code rather than to protect their usage with API level condition checks when they are deprecated.

[8]Generally, the versions are consecutive. However, since Androzoo may miss some versions, sometimes the pair is not formed by strictly consecutive versions.

4.3 RQ3: Practical Usefulness of CiD

While warning on API usages that could be incompatible with some on-device android stacks, CiD provides necessary information to developers for quickly locating, debugging and eventually fixing the reported issues. This includes information about the actual lifetime of the reported API method in the framework, the call chain to reach the flagged API usage, etc. To evaluate whether such information is helpful in practice, we manually submit issue reports to 56 F-Droid projects (one report per project based on the collected warnings.) whose source code is publicly available on Github and contains at least one API-related compatibility issue based on the reports of CiD. Development teams have reacted on 20 out of 56 submitted issue reports[9]. We note that among the 20, six have been systematically closed with a message indicating that "the project is currently abandoned" or without any message that could help assess the usefulness of the issue report. In another case, a close message is formulated as "f-droid, won't fix" suggesting that these developers may feel overwhelmed by irrelevant messages from the F-droid community. Nevertheless, seven of the issue reports are acknowledged and confirmed by developers who have for example tagged them as bugs (cf. details of apps in Table 3). Three of the reports have initiated developer discussions but did not lead to a final decision on fixes. We have noted that several of the bugs are even fixed in a very short time, suggesting that the warnings and details provided by our tool are indeed helpful for developers in order to fix potential API compatibility issues.

After exchanging with developers, we have found that, in the case of three apps, our tool has raised warnings that they consider as false positives. Actually, in two of the cases, CiD is run on the latest app version provided on F-Droid, which happens to be different from the beta-version on Github. We have re-checked with CiD that the version on Github does not present the API compatibility issue raised previously. The other case of false positive is due to the use of *SDK* attribute in condition checks: CiD does not consider it as valid for API level condition checking as it is deprecated since API level 4. To limit related warnings to be treated as false positives, we have updated CiD to take this attribute into account in the construction of the conditional call graph.

Table 3: Experimental Results on the Usefulness of CiD.

App Name	# Commits	Dex Size	Issue ID(s)	Fix Commit
Rabbit Escape	1,785	1,872 K	478	-
ShareViaHttp	167	1,545 K	24	9ed54f
FRCAndroidWidget	285	320 K	32,33	fc0364
Nextcloud Notes	240	4,137 K	177	-
DragonGoApp	180	1,185 K	13	-
EnigmAndroid	54	1,792 K	9	-
Red Screen	6	4 K	2	31a560

RQ3: CiD can provide helpful information to developers for quickly understanding, evaluating and eventually fixing API compatibility issues that their apps may encounter.

[9]We aimed at reporting all identified compatibility issues. Unfortunately, we only managed to successfully report 56 issues because our GitHub account was banned due to spam suspicions.

4.4 RQ4: Comparison with State-of-the-art

To the best of our knowledge, there is no state-of-the-art work that specifically focuses on taming Android API compatibility issues. FicFinder [9] is the closest work to ours on detecting compatibility issues. We thus evaluate the performance of FicFinder and CiD against two datasets for comparison purpose: **7 benchmark apps** that are developed internally within our team to cover the variety of challenges mentioned in Section 3 and **1,797 real-world apps** that are collected from F-Droid. We remind the readers that the objective of FicFinder is to solve Android fragmentation-induced compatibilities issues that go beyond API compatibility issues by also pinpointing other compatibility issues such as the device-specific ones, which are however not the focus of our approach.

Benchmark Apps. Table 4 lists all the seven benchmark apps and provides comparative results between FicFinder and CiD. Overall, CiD correctly resolves all the compatibility issues while FicFinder analyzes correctly only two app cases: 1) *Basic*, where the issue-induced API (i.e., *setExact*) is modeled by FicFinder and 2) *Protection*, where no issue is reported, which is expected as the issue-induced API is accessed with protection. Except for these two cases, FicFinder reports one false positive and 6 false negative results. The false positive is reported because FicFinder does not consider the case when constructor methods are involved in protecting the access of APIs which is not available in all versions. The false negatives are reported because the database of templates provided by FicFinder at the moment is very small, and does not contain models for the API methods used in these benchmark apps. It is also worth to mention that FicFinder is not specifically designed for taming API compatibility issues and its database is built through a summary of fixed compatibility issues, it is not systematic and thus will likely miss some relevant APIs. Our API lifecycle modelling (ALM) module can eventually be leveraged by FicFinder to enrich its database and thus cover more compatibility cases in a systematic and regular way.

Table 4: Comparison results of FicFinder and CiD on Benchmark apps.

⍟ = correct warning, ★ = false warning, ○ = missed issue

App	Relevant APIs	FicFinder	CiD
Basic	AlarmManager.setExact	⍟	⍟
Generic Type	TreeMap.replace	○	⍟
Varargs	KeyProtection.Builder.<init> KeyProtection.Builder.setBlockModes	○ ○	⍟ ⍟
Inheritance	TransitionManager.go Activity.getContentTransitionManager	○ ○	⍟ ⍟
Forward	AssetInputStream.getAssetInt	○	⍟
ConditionCheck	AlarmManager.setExact		
ConditionCheck2	AlarmManager.setExact	★	

Real-World Apps. We focus on backward compatibility issues as they are specifically considered by FicFinder. For the 1,797 F-Droid apps, FicFinder reports in total 603 compatibility issues for 214 apps, while as discussed previously CiD reports 9,755 issues for 609 apps. The median number of reported issues are 0 and 3 respectively by FicFinder and CiD. We confirmed with the MWW test that this difference is statistically significant at a significance level of 0.001. We further manually check the different cases of mismatched results and find that all the backward compatibility issues identified by FicFinder are also identified by CiD, except for 15 apps. In all the 15 apps, compatibility issues with a single API, namely *openDatabase()* of class *SQLiteDatabase*, seems to be missed out by CiD. Indeed, while FicFinder reports that *openDatabase()* is introduced only from API level 11, the ALM module has modelled its lifetime as starting since API level 1. We then refer to the Android developer documentation [23] which confirms that *openDatabase()* is indeed introduced since API level 1. These false positives by FicFinder further highlight the limitations of building the approach from manual inputs (in this case issue reports).

RQ4: CiD outperforms the state-of-the-art both on benchmark apps (where it shows its capability to take into account various challenges) and on real-world apps (where it shows its capability to avoid false positives while subsuming all detection results of FicFinder).

5 DISCUSSION

5.1 Threats To Validity

First, to generate the Android API lifecycle model, we have focused on a subset of Android releases. It is possible that we missed some cases of API evolution between non-considered releases. However, to alleviate this threat, we have considered a release in each API level. As suggested by Android documentation, the API level is incremented only when important changes are performed on the API.

Second, our analysis in the API usage extraction may not be very precise. Indeed, the conditional call graph (CCG) that we build in this work is not context-sensitive. We have further made some approximations to integrate constructor methods into the CCG, attempting to be the most conservative so as not to miss potential compatibility issues.

Third, our evaluation results presented in RQ2 could be threatened by the fact that an API is removed/replaced during a consecutive app update may not always be the case of compatibility fixes (e.g., because of functionality dropping). Nevertheless, our evaluation is conducted on a large set of apps, where a small number of false alarms would not impact the final statistic results.

Fourth, for forward compatibility warnings, at the moment, CiD is focused on identifying Android APIs that are removed from the public SDK. The behaviour of such now-hidden APIs (i.e., same signature but different implementation) may also evolve and contradict with developer initial expectations. In future work, we plan to take this research direction for further enhancing CiD.

Finally, several aspects of our assessment have involved manual efforts, e.g., to check the reported results or to submit issue reports. Thus, we cannot guarantee that no error was made as manual tasks are subject to bias and mistakes. To mitigate this threat, we have cross-validated our results and ensured consistency.

5.2 Lessons Learned

Our study on the use of *minSdkVersion* exposes a contradiction in the behaviour of app developers who want to use interesting, but potentially problematic APIs while targeting a large range of devices. Thus, we have noted that most recent API methods, which provide up-to-date and performant functionalities in higher API levels, are used in several apps where developers still keep a small *minSdkVersion* value to increase the number of potential users, including those using platforms with lower API levels. In other words, for some developers, maximizing the number of targeted users seems to be more important than the reliability of their apps.

This study also highlights a significant problem in enforcing documentation recommendations. We have indeed found in a sample set of apps from the Google Play official markets that several app developers do not set the *minSdkVersion* attribute although it is recommended, while others set the *maxSdkVersion* attribute although it is not recommended.

Overall, our investigations suggest that market maintainers could benefit from tools such as CiD to restrict the propagation of apps that could crash on users devices. Indeed, developers cannot be trusted to follow recommendations, and market reputation[10] can be at stake when thorough verifications are not enforced.

6 RELATED WORK

The high fragmentation in the Android ecosystem (with a variety of device brands running even more diverse platform versions) is challenging developers for exhaustively testing their apps to expose API compatibility issues. Recent studies have explored API evolution as well as associated compatibility problems in several aspects. We summarize important related work in this section.

API Evolution. API evolution is an important aspect of software maintenance [24, 25]. Similar to our work, Li et al. [8] have mined different releases of the Android framework to investigate the evolution of Android APIs. They, however, are interested in inaccessible (a.k.a, internal/hidden) Android APIs. In this work, we focus on public APIs used by developers in their apps. Since some public APIs may become inaccessible and cause compatibility issues (with app crashes), some of our findings are also in line with those reported by their work [8]. Furthermore, as revealed by Li et al. [26], the fact that APIs leveraged by common and ad libraries are not well updated may introduce security and maintenance problems to the apps [27–29].

Linares-Vásquez et al. have shown that the evolution of Android platforms could impact app ratings [30, 31]. In particular, they have shown that more successful Android apps generally use less change-prone APIs, which if used would likely to initiate stack overflow discussions [32]. More questions and discussions could be induced if the API's behaviour is further changed (i.e., the method's body is massively modified). McDonnell et al. [33] have conducted an investigation on the stability and adoption of Android APIs, in which they demonstrate that Android is evolving fast at a rate of 115 API updates per month on average, while the average time taken by developers to adopt new versions is much longer comparing to the fast-evolving APIs.

API evolution in other platforms has also been extensively studied in the literature. For instance, Businge et al. have conducted several empirical studies [34–36] on the utilization of unstable APIs from the Eclipse platform. Hora et al. have also investigated how developers react to API evolution for the Pharo system [37]. These findings show that API evolution can have a large impact on a software ecosystem in terms of client systems, methods, and developers. Overall, API evolution is quite common in big systems and is a source of compatibility issues.

[10]Particularly, in regions such as China where there is no hegemony of a single market.

Compatibility Issues. Android fragmentation is a well documented source of various problems [38–43]. Liu et al. have found that Android performance bugs could be detected only when testing some specific devices and Android platforms [44], while Pathak et al. have demonstrated that the recurrent Android OS updates have caused a large fraction of user complaints about energy bugs [33]. In a 2012 usability study, Nayebi et al. have found that the different resolutions of device displays challenge the design and implementation of Android apps, leading to serious compatibility issues [45]. Our work is complementary to these related works by focusing on API-related compatibility issues.

The closest work to ours for addressing API-induced compatibility issues is FicFinder [9]. Wei. et al. have recently proposed this tool for Android developers. Their work, however, differs from ours in several ways: 1) The database (i.e., API-context pairs) leveraged by FicFinder to flag compatibility issues is built through a manual process, resulting in a small database which will likely lead to a high rate of false negatives (i.e., missing real compatibility issues). Our CiD approach is based on an API lifecycle model which is built systematically and automatically by mining changes between different updates of the Android framework base. 2) Our work focuses mainly on API compatibility issues, including both forward and backward compatibility. FicFinder, by mining issue reports, is modelling other device-specific issues.

7 CONCLUSION

We have contributed in this paper with CiD, an approach for automating the detection of API-related compatibility issues in Android apps. Our approach builds mainly in three steps where we first analyse the history of framework releases to model the lifecycle of Android API methods. Then, we build a static analyser for pinpointing app code locations where an API method is used with no API level checking when it could be running on mismatched platform versions. Last, we combine the outputs of the previous two steps to flag potential compatibility issues. Our assessment of CiD on both open source and close source apps shows that our approach is effective in detecting compatibility issues. We further show that CiD provides enough information to confirm the problem to developers who could then quickly apply relevant fixes. Finally, we demonstrate that CiD outperforms the recent state-of-the-art approach for detecting API-related compatibility issues. Our future work will mainly focus on forward compatibility issues to warn on behaviour changes in API that are only hidden to support, to some extent, forward compatibility for non-maintained apps.

ACKNOWLEDGMENTS

This work was partially supported by the Fonds National de la Recherche (FNR) of Luxembourg under project RECOMMEND C15/IS/10449467 and CHARACTERIZE C17/IS/11693861, and the BUPT Youth Research and Innovation Program (No. 2017RC40), the National Natural Science Foundation of China (No. 61702045), the National Key Research and Development Program of China (No.2017YFB0801903).

REFERENCES

[1] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[2] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *The 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC)*, 2015.

[3] Android platform frameworks base. https://github.com/android/platform_frameworks_base. Accessed: 2017-02-10.

[4] Issue 225647: Nosuchmethoderror in forwardinglistener. https://code.google.com/p/android/issues/detail?id=225647. Accessed: 2017-02-10.

[5] java.lang.nosuchmethoderror: android.graphics.canvas.drawoval#22. https://github.com/googlesamples/android-vision/issues/22. Accessed: 2017-02-10.

[6] Haoyu Wang, Zhe Liu, Yao Guo, Xiangqun Chen, Miao Zhang, Guoai Xu, and Jason Hong. An explorative study of the mobile app ecosystem from app developers' perspective. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 163–172, 2017.

[7] F-droid âĂŞ free and open source android app repository. https://f-droid.org. Accessed: 2017-02-10.

[8] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

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

[10] Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John Mitchell. Target fragmentation in android apps. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 204–213. IEEE, 2016.

[11] Tao Zhang, Jerry Gao, Jing Cheng, and Tadahiro Uehara. Compatibility testing service for mobile applications. In *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*, pages 179–186. IEEE, 2015.

[12] Hyung Kil Ham and Young Bom Park. Mobile application compatibility test system design for android fragmentation. In *International Conference on Advanced Software Engineering and Its Applications*, pages 314–320. Springer, 2011.

[13] Android version history. https://en.wikipedia.org/wiki/Android_version_history. Accessed: 2017-02-10.

[14] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 33–44. IEEE, 2016.

[15] Android maxsdkversion. https://developer.android.com/guide/topics/manifest/uses-sdk-element.html. Accessed: 2017-02-10.

[16] Application forward compatibility. https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#fc. Accessed: 2017-02-10.

[17] Application backward compatibility. https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#bc. Accessed: 2017-02-10.

[18] Getting mac address in android 6.0. http://stackoverflow.com/questions/33159224/getting-mac-address-in-android-6-0. Accessed: 2017-02-10.

[19] Yasfa1: Yet another simple forms app. https://github.com/IanEH/YASFA1. Accessed: 2017-02-10.

[20] Li Li, Tegawendé Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *ISSTA*, 2016.

[21] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

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

[23] Android developers: Opendatabase. https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html#openDatabase(java.lang.String,%20android.database.sqlite.SQLiteDatabase.CursorFactory,%20int). Accessed: 2017-02-10.

[24] Meiyappan Nagappan and Emad Shihab. Future trends in software engineering research for mobile apps. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 21–32. IEEE, 2016.

[25] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 2016.

[26] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[27] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[28] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016.

[29] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[30] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change-and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2015.

[31] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

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

[33] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[34] John Businge, Alexander Serebrenik, and Mark van den Brand. Survival of eclipse third-party plug-ins. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 368–377. IEEE, 2012.

[35] John Businge, Alexander Serebrenik, and Mark van den Brand. Analyzing the eclipse api usage: Putting the developer in the loop. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 37–46. IEEE, 2013.

[36] John Businge, Alexander Serebrenik, and Mark GJ van den Brand. Eclipse api usage: the good and the bad. *Software Quality Journal*, 23(1):107–141, 2015.

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

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

[39] Huoran Li, Xuan Lu, Xuanzhe Liu, Tao Xie, Kaigui Bian, Felix Xiaozhu Lin, Qiaozhu Mei, and Feng Feng. Characterizing smartphone usage patterns from millions of android users. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 459–472. ACM, 2015.

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

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

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

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

[44] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[45] Fatih Nayebi, Jean-Marc Desharnais, and Alain Abran. The state of the art of mobile application usability evaluation. In *Electrical & Computer Engineering (CCECE), 2012 25th IEEE Canadian Conference on*, pages 1–4. IEEE, 2012.