Understanding and Detecting Evolution-Induced Compatibility Issues in Android Apps

Dongjie He
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
hedongjie@ict.ac.cn

Lian Li*
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
lianli@ict.ac.cn

Lei Wang
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
wanglei2011@ict.ac.cn

Hengjie Zheng
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
zhenghengjie@ict.ac.cn

Guangwei Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
liguangwei@ict.ac.cn

Jingling Xue
University of New South Wales
School of Computer Science and Engineering
Sydney, Australia
jingling@cse.unsw.edu.au

ABSTRACT

The frequent release of Android OS and its various versions bring many compatibility issues to Android Apps. This paper studies and addresses such evolution-induced compatibility problems. We conduct an extensive empirical study over 11 different Android versions and 4,936 Android Apps. Our study shows that there are drastic API changes between adjacent Android versions, with averagely 140.8 new types, 1,505.6 new methods, and 979.2 new fields being introduced in each release. However, the Android Support Library (provided by the Android OS) only supports less than 23% of the newly added methods, with much less support for new types and fields. As a result, 91.84% of Android Apps write additional code to support different OS versions. Furthermore, 88.65% of the supporting codes share a common pattern, which directly compares variable `android.os.Build.VERSION.SDK_INT` with a constant version number, to use an API of particular versions.

Based on our findings, we develop a new tool called IctApiFinder, to detect incompatible API usages in Android applications. IctApiFinder effectively computes the OS versions on which an API may be invoked, using an inter-procedural data-flow analysis framework. It detects numerous incompatible API usages in 361 out of 1,425 Apps. Compared to Android Lint, IctApiFinder is sound and able to reduce the false positives by 82.1%. We have reported the issues to 13 Apps developers. At present, 5 of them have already been confirmed by the original developers and 3 of them have already been fixed.

*Corresponding author

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238185

CCS CONCEPTS

• **Software and its engineering** → *Automated static analysis*; **Software reliability**; *Software safety*;

KEYWORDS

Android compatibility, incompatible API usage, Android evolution

ACM Reference Format:
Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and Detecting Evolution-Induced Compatibility Issues in Android Apps. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238185

1 INTRODUCTION

Android is the most popular mobile operating system with over 80% market share [10]. The number of Android applications is increasing at an alarming speed, with about 35,000 new Apps released on Google Play every month [1]. However, Android OS is released frequently and it is a well-known challenge for the application developers to deal with compatibility issues on different OS versions [27, 39]. This challenge is now a hot topic on internet forums such as Stack Overflow (414 different topics), and the developers have to deal with complaints from users about the poor compatibility of their Apps frequently.

There are no mature tools to detect *evolution-induced compatibility issues* for Android Apps, i.e., compatibility issues caused by Android system evolution. Existing studies have investigated

several aspects of the related issues. For example, previous works [37, 41, 44] try to understand software reuses in Android Apps and find them heavily depend on Android API. McDonnell et al.[36] studied how fast Android API evolves and the impact of API evolution on the compatibility issues of Android Apps. Li et al.[30] studied inaccessible Android APIs and concluded that inaccessible APIs used in Apps are neither forward nor backward compatible. FicFinder[47] uses API-context pairs (manually extracted from known compatibility issues) to detect unknown fragmentation-induced compatibility issues. Although these works are helpful in understanding evolution-induced compatibility issues for Android Apps, little is known on how developers fix such issues, whether these issues are common in Apps and what are their root causes. In addition, existing studies have not investigated these issues down to the source code level. Hence, they cannot provide deeper insights (e.g., common fixing patterns) to understand and mitigate evolution-induced compatibility issues.

To better understand evolution-induced compatibility issues in Android Apps, we conduct an extensive empirical study over the 11 most popular Android OS versions and 4,936 Android Apps. We find that 91.84% of Android Apps write specific code to deal with evolution-induced compatibility issues. This is due to the drastic API changes induced by Android evolution and the insufficient support from the Android Support Library: there are 140.8 new types, 1,505.6 new methods, and 979.2 new fields being introduced in each new SDK release and only 21.60% new types, 22.74% new methods, and 5.36% fields are supported by the Support Library. Furthermore, we find that fixing evolution-induced compatibility issues in Android Apps is usually very simple: 88.65% of them compare the variable `android.os.Build.VERSION.SDK_INT`, abbreviated as `SDK_INT`, with a constant integer value directly to check the versions of the underlying Android OS. We believe these findings can provide guidance to detect, diagnose and fix evolution-induced compatibility issues.

Based on our findings, we develop IctApiFinder, a new tool to automatically detect incompatible API usages in Android Apps. Incompatible API usages are one type of evolution-induced compatibility issues which invoke API methods not supported by the underlying Android versions. They are serious bugs which often crash the Apps and throw "`java.lang.NoSuchMethodError`" exceptions. IctApiFinder computes on which Android versions each API can be invoked, using an inter-procedural data-flow analysis framework. It then checks whether an API invocation is incompatible or not by examining each specific Android SDK version. We have implemented IctApiFinder in Soot[46] and have applied it to 1,425 Android Apps downloaded from F-Droid[8], where 361 Apps have been found to be problematic. We have manually analyzed the bug reports from 20 randomly selected Apps, and have found that our tool could effectively reduce 82.1% false positives compared to Android Lint, a tool available in Android SDK. We have reported our findings to their original developers for 13 of the 20 Apps: 5 reported issues have been acknowledged by their original developers, and 3 of them are considered as critical bugs which have already been fixed. Note that one already-fixed bug is actually caused by an external library, which cannot be found by Android Lint, and is also difficult to diagnose for the developers. To summarize, this paper makes the following contributions:

- We conduct the first empirical study of evolution-induced compatibility issues on large-scale, real-world Android Apps (4,936 Apps and 11 Android OS versions). Our findings can help to better understand and characterize such issues, and shed lights on future studies on this topic.
- We propose a new method to automatically detect incompatible API usages in Android Apps, by precisely computing the *reachable Android OS versions* for each API (the OS versions on which the API may be invoked) using an inter-procedural context-sensitive data-flow analysis framework. Our method drastically improves the precision of existing tools, reducing the false positives of Android Lint by 82.1%.
- We design and implement a new tool, IctApiFinder, to automatically detect incompatible API usages in Android Apps. IctApiFinder have detected incompatible API usage bugs in 361 out of 1,425 Apps. We have reported our findings to their original developers for 13 randomly-selected Apps, 5 reported issues have been acknowledged and 3 critical issues have already been fixed.

The rest of this paper is organized as follows: Section 2 presents the necessary background information. Section 3 describes our empirical study. We propose our detection method in Section 4 and evaluate it in Section 5. We discuss the threats to validity in Section 6 and summarize related works in Section 7. Finally, Section 8 concludes the paper.

2 BACKGROUND

Android is a fast evolving system. The platform provides APIs (i.e., Android SDK) to its applications as the programming interfaces. These interfaces keep changing as Android evolves. By convention, versions of Android SDKs are differentiated using a unique integer identifier, named *API level* [4]. The API level starts from 1, and at present, the API level of the latest SDK version is 27.

2.1 Declare SDK Versions in Android Apps

Listing 1: Example code snippet to declare SDK versions.

```
1 <uses-sdk
2 android:minSdkVersion = "10"
3 android:targetSdkVersion = "27"
4 android:maxSdkVersion = "27" />
```

Android Apps need to declare their supported SDK versions via the `<uses-sdk>` element in their manifest files (i.e., AndroidManifest.xml) [48]. As shown in Listing 1, there are 3 attributes given integer values:

- **`minSdkVersion`.** The `minSdkVersion` value declares the minimum API level supported by an App. The App will not be installed on an Android system if its `minSdkVersion` value is larger than the API level of the underlying system.
- **`targetSdkVersion`.** The `targetSdkVersion` value defines the API level that an App targets at. Android adopts the backward-compatible API behaviors of the declared target SDK version, even when the App is running on a higher SDK version. This design aims to ensure consistent behavior of the Apps on different SDK versions.
- **`maxSdkVersion`.** The `maxSdkVersion` value gives the maximum platform API level on which an App can run. This

attribute is already deprecated since Android 2.1 (API level 7).

The declared SDK versions only suggest on which versions an App can be installed. In practice, App developers commonly use the runtime value of variable `SDK_INT` to check the SDK version of the underlying system [47].

2.2 Android Support Library

Android OS provides the Android Support Library as a basic solution to tackle the increasingly severe evolution-induced compatibility issues. This library was firstly released in 2011. It has since become the most widely used Android library [3]. The Android Support Library consists of a collection of libraries which can roughly be divided into two groups: compatibility and component libraries [15].

Compatibility libraries focus on back porting features for new SDK releases. It provides wrappers for a subset of interfaces (or types) on different SDK versions. Instead of invoking APIs provided by the SDK directly, Apps can call the wrappers in the Support library. As such, Apps developed for a new SDK version may be able to run on previous SDK versions, without modification. The major compatibility libraries are v4- and v7-appcompat.

Component libraries implement features that are not part of the standard framework. These self-contained libraries can be easily added or removed from a project without concerning for dependencies. The major component libraries include v7-recyclerview and v7-cardview. In this paper, we focus on the compatibility libraries since the component libraries do not handle compatibility issues.

2.3 Android Lint

Android Lint is a code scanning tool introduced in ADT (Android Development Tools) 16. It checks for various potential bugs and optimization improvements. The tool integrated in the latest version of Android Studio features more than 200 default checks. One of them called *ApiDetector* aims to detect incompatible API usages. This check scans through all invocations to Android APIs. It warns about an invocation to a particular API if it is not available on SDK versions supported by the App, as declared in its manifest file. Lint ignores code snippets annotated with certain annotations, e.g., `@TargetApi` and `@SuppressLint` [11]. Although not mentioned in any documents, we notice that Lint avoids false positives by ignoring code patterns when an API is invoked in an `if` statement whose condition compares variable `SDK_INT` to an integer value to check the underlying Android SDK version.

3 EMPIRICAL STUDY

The study tries to address the following three research questions.

- **RQ1:(Root cause):** What are the root causes of evolution-induced compatibility issues?
- **RQ2:(Issue severity):** How common are these issues in real Android Apps?
- **RQ3:(Issue fixing):** How do Android developers fix evolution-induced compatibility issues in practice?

3.1 Methodology

To answer the above research questions, we collect a large set of data consisting of 11 Android SDK versions (together with the Android Support Library in these versions), and 4,936 Apps. This subsection presents our datasets and analytical methods.

Table 1: List of selected Android SDK versions.

Level	Revision	Shares	# Types	# Methods	# Fields
16	android-4.1.2_r2.1	1.7%	3,217	30,057	11,679
17	android-4.2.2_r1.2b	2.6%	3,259	30,569	12,004
18	android-4.3_r3.1	0.7%	3,290	31,104	12,512
19	android-4.4_r1.2.0.1	12.0%	3,412	32,139	13,325
21	android-5.0.2_r3	5.4%	3,673	35,426	16,333
22	android-5.1.1_r9	19.2%	3,683	35,568	16,380
23	android-6.0.1_r9	28.1%	3,471	35,239	16,757
24	android-7.0.0_r7	22.3%	3,823	39,773	20,016
25	android-7.1.2_r9	6.2%	3,828	39,896	20,076
26	android-8.0.0_r9	0.8%	4,181	44,307	21,419
27	android-8.1.0_r9	0.3%	4,201	44,455	21,471

3.1.1 Datasets Collection. We consider *API levels* 16-27 in our research. TABLE 1 presents the selected Android SDK versions (Column 2) and their market shares (Column 3). The other versions are not selected since their market shares are negligible. API level 20 is specific to wearable devices thus it is not included in our study either [30]. We compile the sources of these SDK versions downloaded from AOSP [5]. For each version, we extract its SDK (android.jar) and the corresponding Support Library for further study. The last three columns in TABLE 1 give the number of types, methods, and fields in each SDK version, respectively.

We conduct our study using a large set of third-party Apps (in APK format, without source code) downloaded from the AndroZoo repository [18]. AndroZoo is a specialized repository for the research community, and we totally download 8,047 Apps from it. In this study, we only consider Apps targeting our selected API levels (i.e., `targetSdkVersion` value ranges from 16 to 27), and 4,936 Apps are selected.

Table 2: List of manually inspected Apps. The 10 Apps in F-Droid with the most usage counts of variable `SDK_INT` are selected.

APP	Release	KLOC	# SDK_INTs
org.telegram.messenger	4.6.0a	324.2	531
com.poupa.vinylmusicplayer	0.16.4.4	35.8	209
org.glucosio.android	1.3.0-FOSS	8.2	195
com.amaze.filemanager	3.2.1	30.3	185
im.vector.alpha	0.8.1	52.5	185
com.github.axet.maps	8.1.0-4-Google	120.9	179
com.biglybt.android.client	1.1.4	483.8	173
eu.kanade.tachiyomi	0.6.8	2.7	165
org.bottiger.podcast	0.160.2	41.9	165
es.usc.citius.servando.calendula	2.5.3	26.3	154

TABLE 2 lists the 10 open-source Apps downloaded from F-Droid[8] (a popular open-source App store) for manual inspection. The 10 Apps are selected because they frequently use the variable `SDK_INT`, which is commonly used by developers to check specific SDK versions and address compatibility issues on those versions. We write a crawler to download all latest version of the total 1,425 Apps in F-Droid. The 10 Apps which use variable `SDK_INT` for

the most number of times are chosen. Column 4 gives the usage counts of variable SDK_INT for each App. We manually inspect the source codes of the 10 Apps, to understand how developers address evolution-induced compatibility issues in practice.

3.1.2 Analytical Methods. To answer RQ1, we compare the differences between any two adjacent SDK versions. Specifically, we check whether any newly introduced APIs are supported by its corresponding Android Support library or not. To answer RQ2, we use Soot[46] to scan the 4,936 Apps downloaded from AndroZoo, and count how many times the variable SDK_INT is used. In our experiments, we assume that variable SDK_INT is mostly used to test the underlying SDK version and address evolution-induced compatibility issues. We manually inspect the code snippets where SDK_INT is used for the 10 Apps in TABLE 2, to answer RQ3 and validate the above assumption.

3.2 Findings

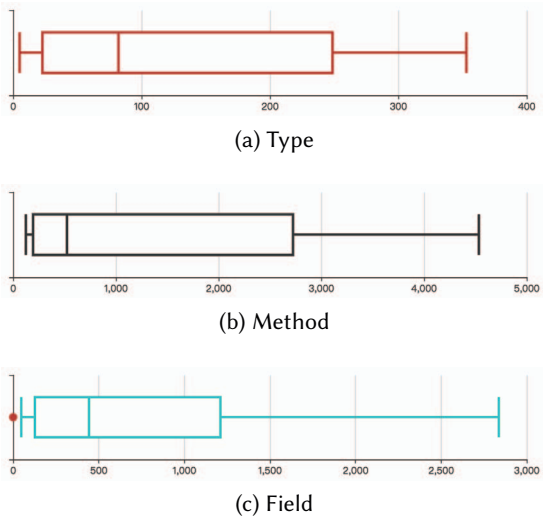

Figure 1: Differences between adjacent SDK versions.

3.2.1 RQ1: Root cause.

- **Finding 1:** Android SDK version evolution leads to significant API changes.

Figure 1 compares the differences between two adjacent SDK versions. We observe dramatic changes as Android SDK evolves. On average, 140.8 new types, 1,505.6 new methods and 979.2 new fields are introduced, as the Android SDK evolves into a new version.

- **Finding 2:** Android Support Library provides support for less than 23% of the new introduced APIs.

The Android Support Library is introduced to ease compatibility issues in the Android ecosystem. We are curious about how well they address compatibility issues between different SDK versions. In our research, we compare any two adjacent SDK versions by checking how many newly introduced APIs (types, methods, and fields) are supported by the Android Support Library. We conservatively assume that an API is supported by the Android Support Library if it is used in the library. This gives us an optimistic estimation, since there are also normal usages besides those as wrapper methods. Disappointingly, the support ratio is only 21.60% for new types, 22.74% for new methods, and 5.36% for new fields, suggesting insufficient support to address the prevalent compatibility issues.

Table 3: API changes supported by the Android Support Library.

	#Supported / #New introduced		
Adjacent Levels	**Type**	**Method**	**Field**
17vs18	0/67	0/744	0/571
18vs19	6/122	75/1,044	91/813
19vs21	11/265	136/3,383	3/3,022
21vs22	0/10	4/154	0/64
22vs23	2/152	2/1,970	0/823
23vs24	102/355	1,100/4,605	179/3,267
24vs25	1/5	7/132	0/60
25vs26	164/357	2,424/4,450	261/1,350

- **Finding 3:** Without considering API behavioral changes, 86% of Apps can directly run on the next Android version without any modification. Thus, evolution-induced compatibility issues are mainly introduced from API behavioral changes and new features in later SDK versions.

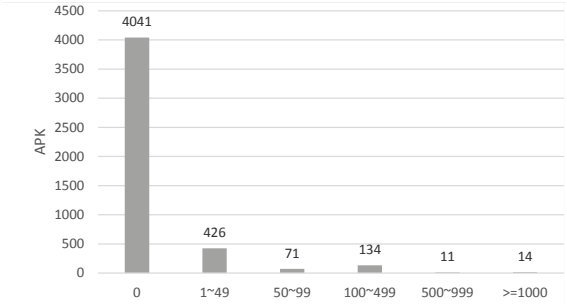

Figure 2: Distribution of Apps using abandoned APIs.

Figure 2 counts how those APIs abandoned in the next SDK version are used in Android Apps. The interesting fact is that 4,041 Apps out of the total 4,697 Apps do not use any abandoned APIs. Hence, if not considering API behavioral changes, we can conclude that 86% Apps can run on a later SDK version without any modification. This implies that developers address evolution-induced compatibility issues mainly because they need to adapt API behavioral changes or use new features in later SDK versions.

Answer to RQ1: To summarize, the main causes of evolution-induced compatibility issues in Android Apps are the drastic API changes induced by Android evolution, and the insufficient support from the Android Support Library. As a result, App developers often have to deal with evolution-induced compatibility issues in order to use latest features and support multiple SDK versions.

3.2.2 RQ2: Issue severity.

- **Finding 4:** 91.84% of Apps write specific code to address evolution-induced compatibility issues.

We use Soot [46] to analyze the 4,936 Apps downloaded from AndroZoo, where 32 Apps cannot be processed. Among the remaining 4,904 Apps, 4,504 Apps use variable `SDK_INT` (usages in the Android support libraries, e.g., classes whose name started with `android.support.*`, are excluded), suggesting that 91.84% of Apps check the underlying Android SDK versions to address evolution-induced compatibility issues in their implementation. We also count how many times `SDK_INT` is used in each App. On average, an App uses variable `SDK_INT` for 55.45 times (Figure 3).

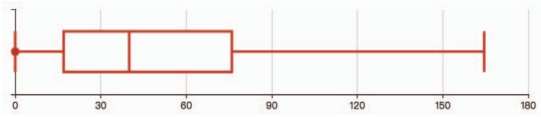

Figure 3: `SDK_INT` usage counts in Apps

- **Finding 5:** Less than 6.74% APIs are frequently used, and `SDK_INT` is the most frequently used field.

Figure 4 studies the usages for each Android API: 54,593 APIs have never been used by the 4,904 Apps we processed, and only 6.74% APIs are used by more than 100 Apps. We have manually inspected the 66 APIs with more than one million usage numbers. They can be classified into three categories: 33 APIs belong to the JDK library, with 18 in `java.lang.*`, 12 in `java.util.*` and 3 in `java.io.*`; 30 APIs start with `android.*`, with 8 in `android.os.Parcel`, 5 in `android.util.Log`, 4 in `android.os.Bundle`, 3 in `android.content.Intent`, 2 in `android.app.Activity` and `android.os.Binder`, and other 6 in 6 diffrent packages, respectively; the remaining 3 APIs all belong to `org.json.JSONObject`. Moreover, we find that `SDK_INT` is the only field with more than one million usage counts, which also confirms our finding that the Apps developers frequently handle evolution-induced compatibility issues by themselves.

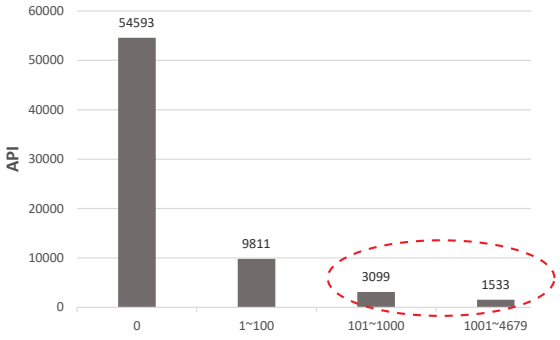

Figure 4: Distribution of APIs by usage counts.

Answer to RQ2: Evolution-induced compatibility issues are very common and about 91.84% of Apps write specific code to deal with such issues.

3.2.3 RQ3: Issue fixing.

- **Finding 6:** most fixing patterns are very simple, complicated patterns are rare.

We have manually inspected the 10 Apps in TABLE 2, and found several common patterns to address evolution-induced compatibility issues. The most common practice is to invoke different APIs directly on different versions, according to the runtime value of `SDK_INT`. For example, Listing 2 shows a code snippet extracted from `com.amaze.filemanager`[2], where the SDK API method `quitSafely`, instead of `quit`, is used after API level 18.

Listing 2: Common practice to address evolution-induced incompatibility issues.

```
1 if (SDK_INT >= 18) {
2     // let it finish up first with what it's doing
3     handlerThread.quitSafely();
4 } else
5     handlerThread.quit();
```

Frequently, the developers introduce wrapper methods to deal with evolution-induced compatibility issues. The code snippet (also extracted from `com.amaze.filemanager`[2]) in Listing 3 invokes different password encrypt wrappers (`CryptUtil.aesEncryptPassword` and `CryptUtil.rsaEncryptPassword`) for different SDK versions, where different SDK APIs are invoked by the wrappers accordingly.

Listing 3: Address evolution-induced compatibility issues using wrapper methods.

```
1 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
2     return CryptUtil.aesEncryptPassword(plainText);
3 } else if (Build.VERSION.SDK_INT >= 18) {
4     return CryptUtil.rsaEncryptPassword(context, plainText);
5 } else
6     return plainText;
```

In addition to directly check the value of `SDK_INT` in an `if` statement, developers sometimes check the value of `SDK_INT` using different forms of expressions, e.g., ternary expressions. Listing 4 uses ternary expressions to decide the frame size for different Android versions.

Listing 4: Address evolution-induced compatibility issues using ternary expression.

```
1 LayoutHelper.createFrame(Build.VERSION.SDK_INT >= 21 ? 56 : 60,
2     Build.VERSION.SDK_INT >= 21 ? 56 : 60, (LocaleController.isRTL
3     ? Gravity.LEFT : Gravity.RIGHT) | Gravity.BOTTOM,
4     LocaleController.isRTL ? 14 : 0, 0, LocaleController.isRTL ? 0 : 14, 14);
```

Complicated patterns are usually applied to adapt a complete new Type. These patterns are rare in real-world Apps (non-exist in the 10 Apps we analyze), but very common in the Android Support Library. For example, `android.support.v4.view.ViewCompat` is a type used to adapt different versions of type `android.view.View`. Listing 5 shows the simplified code snippet. It uses different inner types to wrap the APIs for distinct SDK versions (lines 2-20), then initializes the static instance according to a particular SDK version (lines 21-33).

- **Finding 7:** The most common practice (88.65% of usages) checks the underlying SDK version by comparing the variable `SDK_INT` directly with a constant API level value.

We have manually inspected all usages of variable `SDK_INT` in the 10 Apps in TABLE 2. There are 1,249 usages in total in the source codes of the 10 Apps, usages in external libraries excluded. The usage patterns can be classified into 3 categories: `SDK_INT` is

directly compared with a constant value, and the comparison result is used in conditions of `if` statements (C_1); the value of `SDK_INT` is propagated to other variables appearing in `if` conditions (e.g., via field stores and loads), this is also a complicated pattern involving complex dependencies (C_2); control-flow irrelevant usages like log-printing (C_3).

Listing 5: Complicated fixing strategy in `ViewCompat`.

```
1  public class ViewCompat {
2      interface ViewCompatImpl {
3          void setElevation(View view, float elevation);
4      }
5      static class BaseViewCompatImpl implements ViewCompatImpl {
6          @Override
7          public void setElevation(View view, float elevation) {
8          }
9      }
10     static class EclairMr1ViewCompatImpl extends BaseViewCompatImpl {...}
11     static class GBViewCompatImpl extends EclairMr1ViewCompatImpl {...}
12     ......
13     static class KitKatViewCompatImpl extends JbMr2ViewCompatImpl {...}
14     static class LollipopViewCompatImpl extends KitKatViewCompatImpl {
15         public void setElevation(View view, float elevation) {
16             view.setElevation(elevation);
17         }
18     }
19     ...
20     static class Api24ViewCompatImpl extends MarshmallowViewCompatImpl {...}
21     static final ViewCompatImpl IMPL;
22     static {
23         final int version = android.os.Build.VERSION.SDK_INT;
24         if (BuildCompat.isAtLeastN()) {
25             IMPL = new Api24ViewCompatImpl();
26         } ......
27         else if (version >= 21) {
28             IMPL = new LollipopViewCompatImpl();
29         } ......
30         else {
31             IMPL = new BaseViewCompatImpl();
32         }
33     }
34     public static void setElevation(View view, float elevation) {
35         IMPL.setElevation(view, elevation);
36     }
37 }
```

TABLE 4 gives the usage counts of variable `SDK_INT` by categories. Most of the usages (88.44%) are control flow-related (Columns 2 and 3), i.e., the value of `SDK_INT` is used directly or indirectly in `if` conditions. In addition, most usages (78.4%) adopt the simple common practice C_1 (i.e., directly compare variable `SDK_INT` with a constant value) to check the SDK version of the underlying system. The App `org.telegram.messenger` is an exception, it stores the value of `SDK_INT` to static field `Util.SDK_INT`, which is then checked for 84 times. There are also many statements in that App printing the value of `SDK_INT`. For the other 9 Apps, the percentage of control flow-related usages (Columns 2 and 3), and the percentage of the simple common practice C_1 (Column 2 only) are 97.14% and 95.81%, respectively.

Answer to RQ3: Most evolution-induced compatibility issue fixing patterns are very simple. In particular, the most common practice (78.4%) checks the SDK version of the underlying system by directly comparing the variable `SDK_INT` with a constant value.

Table 4: Categorized Usage counts of variable `SDK_INT`.

App Name	# C_1	# C_2	# C_3
org.telegram.messenger	409	84	109
com.poupa.vinylmusicplayer	48	0	4
org.glucosio.android	5	0	0
com.amaze.filemanager	127	0	1
im.vector.alpha	40	0	3
com.github.axet.maps	39	2	4
com.biglybt.android.client	38	1	1
eu.kanade.tachiyomi	31	0	0
org.bottiger.podcast	80	2	0
es.usc.citius.servando.calendula	27	1	0

4 INCOMPATIBLE API USAGE DETECTION

According to our findings, 91.84% of Apps try to address evolution-induced compatibility issues by checking the underlying SDK version in their implementation. The developers need to use the right version of API on each supported SDK version. However, this process is error-prone and often leads to incompatible API usages. A query on Google and Stack Overflow using the keyword "`Android NoSuchMethodError`" gives us 162,000 results, and 414 topics, respectively (April 17, 2018). It is becoming a prevalent problem. However, there are no tools to detect these issues precisely and effectively. As a result, many Android Apps are poorly tested[28].

Android Lint can be used to detect incompatible API usages. However, it is not commonly used by the developers due to its high false positive rates. Listing 6 gives an example. The API used on line 9 is introduced into SDK after level 11, but the `minSdkVersion` is set to 10. Hence, a "`java.lang.NoSuchMethodError`" exception will be thrown, crashing the App on SDK version 10. For this example, Android Lint will report two issues, on line 9 and on line 12, respectively. The report on line 12 is a false positive because Lint does not apply inter-procedural analysis and does not consider context-sensitivity. In addition, Lint cannot detect incompatible API usages in external libraries, leading to false negatives. Currently, Android development uses Gradle[9] as the automated building tool and Apps rely heavily on external libraries.

Listing 6: Example code snippet with incompatible API usage.

```
1  // minSdkVersion: 10; targetSdkVersion 27.
2  public class MainActivity extends Activity {
3  private TextView mView;
4      protected void onCreate(Bundle bundle) {
5          ...
6          if(Build.VERSION.SDK_INT >= 24)
7              wrapper(mView, c, s, null, i);
8          else
9              mView.startDrag(c, s, null, i); // API1 [11, 23]
10     }
11     private wrapper(View v, ClipData c, ...) {
12         v.startDragAndDrop(c, s, o, i); // API2 [24, 27]
13     }
14 }
```

4.1 Detection Method

We develop a new inter-procedural dataflow analysis to detect incompatible API usages. Definition 4.1 gives the necessary and sufficient conditions for incompatible API usages.

Definition 4.1. For any App, the use of an API is incompatible if and only if it satisfies the following three conditions:

- There exists a SDK version whose API Level is larger than or equal to the declared `minSdkVersion` value of the App.
- The API is used by the App on that SDK version.
- The API is not included in the SDK of that particular version.

It is trivial to check the first and last conditions, as implemented in Android Lint. The challenge lies in how to determine whether an API is used or not on a given SDK version. We formulate this challenge into a classical inter-procedural data-flow analysis, which computes the set of reachable Android versions for each API usage in a context-sensitive manner.

We compute the set of reachable SDK versions at each program point for the App under evaluation. At the entry point, the set includes all SDK versions declared in the manifest file of the APP, e.g., from `minSdkVersion` to the largest level 27. This set is updated at program points checking SDK versions. We consider the common practice where the variable `SDK_INT` is directly compared to a constant integer value (Finding 7), and the comparison result is used as conditions of `if` statements. These `if` statements are referred to as *checkpoint statements*. The set of reachable SDK versions in the true or false branch of the checkpoint statement are updated accordingly. Equations 1-2 give the data flow functions, where $CHKED_i$ is defined according to the condition of checkpoint statements. For example, if the condition is $SDK\_INT \leq 24$, then $CHKED_i = \{1, 2, \cdots, 23\}$ (assuming `minSdkVersion` is 1) and $\overline{CHKED_i} = \{24, 25, 26, 27\}$.

$$IN_i = \bigcup_{p \in pred_i} (OUT_p) \tag{1}$$

$$OUT_i = \begin{cases} IN_i \cap CHKED_i & \text{true branch of checkpoint statement} \\ IN_i \cap \overline{CHKED_i} & \text{false branch of checkpoint statement} \\ IN_i & \text{otherwise.} \end{cases} \tag{2}$$

For each usage of API_i, we check whether API_i is included in any reachable SDK version at the usage point or not. If not, a bug is reported.

4.2 Implementation

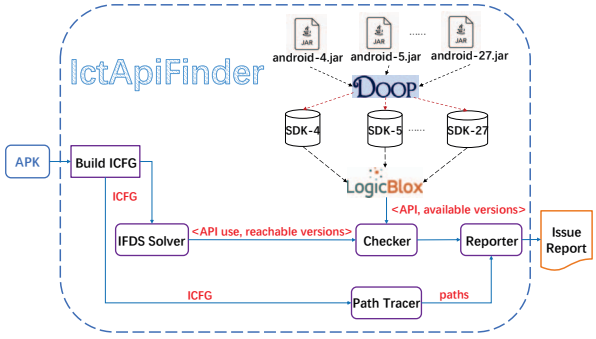

Figure 5: The Architecture of IctApiFinder.

We implement IctApiFinder (InCompaTible API usage Finder) in Soot. The tool detects incompatible API usages in Android Apps by analyzing the .apk file of an App directly.

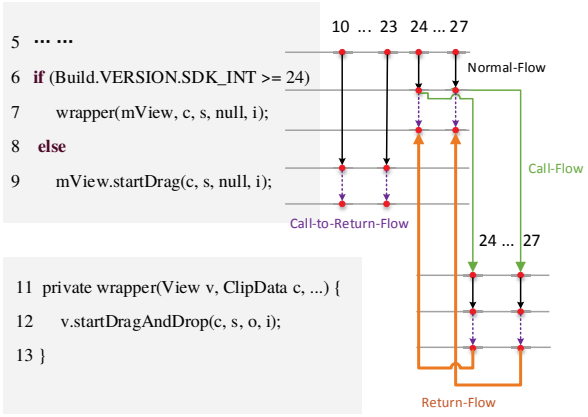

Figure 6: Illustration example to detect incompatible API usages in Listing 6.

Figure 5 depicts the architecture of our implementation. We build the inter-procedural control flow graph (ICFG) for Android Apps using Soot's SPARK[29] call graph construction algorithm [19]. The inter-procedural data flow solver is implemented on top of Heros[22], a commonly used IFDS framework[40]. To check whether a given API is included in a particular SDK version or not, we use Doop[42], a framework for points-to analysis of Java programs, to extract APIs from SDK (android.jar) file and use a datalog engine, LogicBlox[17, 23] to load API information for each SDK version. Previous works[36][48] extract such information from a SDK document called api-versions.xml, which is not as accurate [48].

Next, we give a brief description on how the IFDS framework computes reachable SDK versions at each program point, and how we detect incompatible API usages using an example.

4.2.1 IFDS Framework. IFDS is a classical context-sensitive inter-procedural data flow analysis framework. This framework can be used to find precise solutions to a general class of inter-procedural data-flow-analysis problems, where the set of data flow facts D is a finite set and the data flow functions are distributive.

The IFDS framework formulates the dataflow analysis problem into a general graph reachability problem on a supergraph extended from ICFG. Nodes are elements in the finite domain of data-flow facts, and edges encode the semantics of transferring functions. There are four types of edges: normal-flow edges to propagate data-flow facts within a procedural; and call-flow edges, return-flow edges, and call-to-return-flow edges to propagate data-flow facts inter-procedurally. As shown in Figure 6, for each program point in the ICFG, there is a set of nodes in the extended supergraph, where each node represents an SDK version number at a program point. Edges connect nodes representing the same SDK version number at successive program points, to propagate the reachable SDK version to the next program point. At a checkpoint statement C (line 6), an edge from a node before C and after C exists only if its corresponding SDK version number satisfies the checked condition. An SDK version is reachable at a program point if there exists a path from the entry to its corresponding node of the SDK version at that program point.

4.2.2 Detection Example. Figure 6 shows how we detect incompatible API usages in the example in Listing 6. At the program point before line 6, the reachable SDK versions are $\{10, \cdots, 27\}$, as declared in the manifest file. Line 6 is a checkpoint statement. According to Equation 2, the reachable SDK versions at line 7 and line 9 are $\{24, \cdots, 27\}$ and $\{10, \cdots, 23\}$, respectively. Line 7 invokes the `wrapper` method. So the reachable SDK versions at line 7, $\{24, \cdots, 27\}$, are propagated to line 12, along the call-flow edges.

At the checking stage, IctApiFinder does not report the false positive in line 12 since the API used on line 12 exists in SDK versions 24-27 according to the extracted information. However, the API used on line 9 does not exist in SDK 10. Hence, IctApiFinder reports an incompatible API usage bug on line 9.

5 EVALUATION

In this section, we evaluate IctApiFinder using the total 1,426 real-world open source Android Apps from F-Droid (a popular open-source App store). We do not test with the 4,936 Apps from AndroZoo since it will be difficult to verify the results without source code information. All experiments are conducted on an Intel(R) Core(TM) i5-4590 box with 4 CPU cores and 16GB memory. The underlying OS is Ubuntu 16.04.4 LTS. Our evaluation aims to answer the following two research questions:

- **RQ4: precision of IctApiFinder :** Can IctApiFinder provide more precise detection results for Apps developers?
- **RQ5: usefulness of IctApiFinder:** Can IctApiFinder help to detect unknown incompatible API usages in real-world Android Apps? Can it provide useful information for Apps developers to diagnose and fix incompatible API usage issues?

5.1 RQ4: precision of IctApiFinder

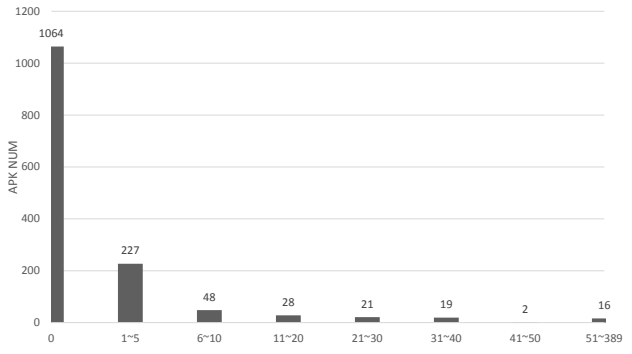

Figure 7: Distribution of Apps in F-Droid by incompatible API usage counts

We apply IctApiFinder to all Apps (using the latest version) available in F-Droid[8], i.e., the total 1,426 Apps in F-Droid. The App `pl.hypeapp.endoscope_5` cannot be processed by Soot [46]. On average, our tool processes an App in 6.08 seconds. The most time-consuming App is `com.nextcloud.client_30000399`, which takes 3 mintues and 45 seconds to analyze. Figure 7 shows the number of incompatible API usages reported by our tool in the total 1,425 Apps we analyzed. IctApiFinder finds incompatible API usages in 361 (25.33%) of the total 1,425 Apps. Although the Apps developers have made extensive efforts to address compatibility issues, many Apps still suffer from incompatible API usages.

We randomly select 20 out of the 361 Apps with incompatible API usage issues for manual inspection. TABLE 5 lists the 20 Apps we choose. The '`APP`' and '`Version`' columns give their names and versions, respectively. Column 4 and 5 present the number of incompatible API usages reported by Android Lint and IctApiFinder, respectively. IctApiFinder is sound in reporting incompatible API usages because of an over-approximate strategy used by the IFDS solver. Hence, it will never miss any incompatible API usage bugs. However, Android Lint often suffers from false negatives because it will skip processing sources with certain annotations (e.g. `@SuppressLint`, `@TargetApi`). In this experiment, we remove these annotation tags for a fair comparison. In addition, Lint does not process libraries thus often misses incompatible API usage bugs in external libraries. To minimize the effect of such kind of false negatives, we conservatively add all issues reported by IctApiFinder in external libraries into that of Lint.

By comparing Column **# Lint** and **# IctApiFinder**, we find that the issues reported by IctApiFinder are significantly less than that of Lint. On average, IctApiFinder can effectively reduce the false positive rate of Android Lint by 82.1%.

> **Answer to RQ4:** In conclusion, IctApiFinder largely reduces the false positive rate of Android Lint by 82.1%. It processes an App within 7 seconds on average.

5.2 RQ5: Usefulness of IctApiFinder

We have manually checked all reports generated by IctApiFinder for the 20 Apps in TABLE 5. Columns **# TP** and **# FP** present the results. IctApiFinder reports 217 issues in the 20 Apps, including 71 false positives, with a false positive rate of 32.72%. Most of the false positives are due to imprecision in the inter-procedural control-flow graph: our algorithm soundly assumes that all components in an APK are directly reachable from the entry without considering the complicated conditions to trigger a component.

After manual inspection, we believe that 13 Apps suffer from real incompatible API usages and have reported them to their original developers. At present, we have received confirmation from the developers of the 4 Apps: `com.vonglasow.michael.qz` (the 6th App), `com.xargsgrep.portknocker` (the 7th App), `com.zegoggles.smssync` (the 9th App), and `org.severalproject` (the 16th App). These issues are color-flagged in red in Table 5. The 3 issues we reported in the App `com.zegoggles.smssync` (the 9th App) are actually false positives since it applies a complicated strategy to address incompatible API usages, which is not considered in our current implementation. For the App `it.feio.android.omninotes.foss` (the 12th App), although we did not receive any confirmation from its developers directly, the developers have added a '`Development`' tag to our reports in their issue tracking system, suggesting further action needed. These issues are color-flagged in orange in TABLE 5. The issue in `jonas.tool.saveForOffline` (the 14th App) is color-flagged

Table 5: Effectiveness of IctApiFinder over 20 randomly-selected Apps

ID	APP	Version	# Lint	# IctApiFinder	# TP	# FP
1	com.github.premnirmal.tickerwidget	2.4.04	17	3	3	0
2	de.christinecoenen.code.zapp	1.10.0	21	1	0	1
3	ca.rmen.android.networkmonitor	1.30.0	46	13	12	1
4	com.easytarget.micopi	3.6.11	2	1	0	1
5	com.prhlt.aemus.Read4SpeechExperiments	1.1	1	1	0	1
6	com.vonglasow.michael.qz	1.1	32	7	7	0
7	com.xargsgrep.portknocker	1.0.11	44	17	13	4
8	com.ymber.eleven	1.0	15	9	9	0
9	com.zegoggles.smssync	1.5.11-beta7	5	3	0	3
10	de.devmil.muzei.bingimageofthedayartsource	1.4	37	37	37	0
11	de.kromke.andreas.unpopmusicplayerfree	1.41	29	14	0	14
12	it.feio.android.omninotes.foss	5.4.3	37	28	24	4
13	jackpal.androidterm	1.0.70-rebuild	52	14	0	14
14	jonas.tool.saveForOffline	3.1.6	3	1	1	0
15	net.opendasharchive.openarchive.release	0.0.17-alpha-1	12	8	0	8
16	org.servalproject	0.93	5	1	1	0
17	org.openintents.notepad	1.5.4	4	3	2	1
18	org.sensors2.osc	0.2.0	25	14	0	14
19	org.smssecure.smssecure	0.16.8-unstable	93	5	2	3
20	org.softeg.slartus.forpdaplus	3.4.8.2	732	37	35	2

in green because we can successfully trigger this bug and crash the App.

In the following, we discuss some real incompatible API usages detected by our tool.

5.2.1 jonas.tool.saveForOffline. This App[13] (the 14th App) downloads web pages for off-line reading. Its `minSdkVersion` value is 16. The App invokes the API `android.webkit.WebSettings.setMediaPlaybackRequiresUserGesture`, which is introduced into SDK after version 17. We run this App on a GALAXY S3 (*API level* 16) device rented from WeTest[16]. The App directly crashed while browsing off-line pages and threw a "`java.lang.NoSuchMethodError`" exception.

5.2.2 com.xargsgrep.portknocker. This App[12] (the 7th App) is a basic port knocker client and its `minSdkVersion` value is 10. It uses the external component `com.ianhanniballake.localstorage.LocalStorageProvider`, which is inherited from the API `android.provider.DocumentsProvider`. However, this API is introduced since SDK version 19. We reported the issues to the original developers and they confirmed them in a 2 days. The issues are fixed in revision 7f37522[7] by increasing the App's `minSdkVersion` to 19. This kind of incompatible API usages are very common since third party-libraries are frequently used in Apps. It is also very difficult to avoid by the developers. Note that Android Lint does not process external libraries. IctApiFinder successfully finds these incompatible API usage issues, demonstrating its effectiveness.

5.2.3 org.servalproject. This App (the 16th App, also called `Batphone`)[14] provides free and secure phone-to-phone voice calling, SMS and file sharing over Wi-Fi, without the need for a SIM card or a commercial mobile telephone carrier. This App's `minSdkVersion` value is 8 while it uses the API `java.lang.String: void String(byte[], int,int,java.nio.charset.Charset)` which is added into SDK since level 9. The developers thanked us and fixed this issue in revision 05e784a[6] by using `java.lang.String: void String(byte[],int, int,java.lang.String)` on SDK version 8.

The above examples show that IctApiFinder can detect critical unknown incompatible API usage issues in real-world Android Apps, including these issues deeply hidden in external libraries. Such issues are very common, but hard to be detected by the developers and Android Lint.

To help with bug diagnosis and verification, we also implement a path tracer which provides up to 10 possible reachable paths to the developers for each incompatible API usage. Our bug reports present the API usage, the incompatible versions, as well as the reachable paths which could help the developers to quickly diagnose and fix incompatible API usages.

> **Answer to RQ5:** IctApiFinder is useful in detecting unknown incompatible API usages. We have found numerous real incompatible API usages in 13 of the 20 Apps manually inspected, where issues reported in 5 Apps have already been confirmed or directly triggered. It also demonstrates its effectiveness by reporting incompatible API usages in external libraries, which are common but difficult to find by developers and Android Lint. The report of IctApiFinder includes detailed information such as reachable paths and incompatible versions, which is helpful for developers to quickly diagnose and fix the reported issues.

6 DISCUSSIONS

6.1 Threats To Validity

Subject selection. The validity of our empirical study results may be subject to the threat that we only manually inspect 10 Android Apps as subjects in analyzing evolution-induced compatibility issues fixing patterns. However, these 10 Apps are selected from 1,425 candidate Apps from F-Droid as they contain the most number of fixing practices to address evolution-induced compatibility issues, with a total number of 1,249 usages of the variable `SDK_INT`. More importantly, the findings obtained from studying these 10 Apps have been proven to be useful in detecting unknown incompatible API usages in real-world Apps.

Errors in manual inspection. Our study may suffer from errors in manually analyzing code snippets which uses variable `SDK_INT` to address evolution-induced compatibility issues. To reduce this

threat, we follow the widely-adopted cross-validating method to ensure the correctness of our results.

Assumptions. In our empirical study, we make two assumptions. The first assumption is that usages of variable `SDK_INT` all address evolution-induced compatibility issues. In practice, there also exist other usages such as log-printing. However, these usages only account for less than 11.56% of the total usages in our study. Another assumption is that a new API is supported by the Android Support Library if it is used in the Support Library. This is a conservative assumption since there are also normal usages of the API. The API support ratio provided by the Android Support Library will be even lower. Hence it does not affect our conclusion that the support from the Android Support Library is insufficient.

Android OS Evolution. The last threat may come from the strategy of Android evolution. All our empirical findings are based on current major android versions. However, Android is a fast evolving system and many OS versions will be gradually phased out. There may be significant changes in the Android ecosystem to address evolution-induced compatibility issues. We cannot guarantee that our findings still hold in the remote future.

6.2 Further Reduce False Positives

In our empirical study, we have classified the usages of `SDK_INT` into three categories. IctApiFinder only considers the most common practice (C_1). There are also complicated cases which require precise pointer analysis [32, 33, 43, 45] to track dependencies of variable `SDK_INT`, or complicated fixing strategies as in List 5. We plan to address these issues in our future work.

Most of the false positives are due to the imprecision of our inter-procedural control flow graph (ICFG). Currently, the ICFG we use is actually same as the one in Flowdroid [20], which conservatively assumes that all components in Android Apps are directly reachable from the entry point, without considering the complex control flows to trigger a component. However, this is not true. For example, some "`Activity`" can only be reached after the call to "`startActivityForResult`". Hence, a more precise ICFG which is required to further reduce false positives. In general, it requires control flow specialization [26, 51] and reflection analysis [50] to build the precise ICFG.

7 RELATED WORK

To the best of our knowledge, we are the first to quantify Android evolution-induced compatibility issues with data from a large body of real Android Apps and provide tool to detect these issues. Existing work have studied the general Android API evolution problem and fragment-induced compatibility issues.

Android API evolution. The maintenance of mobile applications remains to be largely undiscovered in the software maintenance field[38]. API evolution is a frequently research topic in this area. McDonnell et al.[36] have performed an empirical study on API stability and adoption in Android, in which they showed that Android is rapidly evolving, at a rate of 115 averagely API updates per month. However, compared to the fast evolving APIs, it takes much longer time on average to adopt new versions in Android Apps. Linares et al. [35] have shown that Android API changes will trigger more Stack Overflow discussions. Work [34] and [21] investigated the relationship between the popularity of Android Apps and the SDK API changes. Their empirical study pointed out that more popular Android Apps generally tend to use APIs that are less change-prone. The above works are helpful in learning evolution-induced compatibility issues. This paper extends existing works by showing the root causes and quantifying the severity of evolution-induced compatibility issues in real Android Apps. Our findings facilitate effective detection and diagnose of evolution-induced compatibility issues in practice.

Android compatibility issues. Android fragmentation also causes portability and compatibility issues within the entire Android ecosystem [25, 49, 52]. A few recent works have been trying to address these fragmentation-induced compatibility issues. Ham et al.[24] proposed a Device API Level Check Method. Their method records the test results of Android API for each device in a pre-stored database, which is then used to check API usage information and detect compatibility issues. Wei et al.[47] manually extract API usage information (referred to as API-Context pair) from existing compatibility issues, and use such information to detect fragmentation-induced compatibility issues. The latest work, CiD [31], detects evolution-induced compatibility issues by building a so-called conditional call graph, which is not context-sensitive. This paper targets evolution-induced compatibility issues, and we apply a context-sensitive data-flow analysis to automatically detect incompatibility issues, without manual annotation.

8 CONCLUSION AND FUTURE WORK

This paper conducts an extensive empirical study on evolution-induced compatibility issues in Android Apps. Our studies discover the following interesting findings: the Android Support library provides support for less than 23% of the new APIs in each release, and most Apps (91.84%) need to address evolution-induced compatibility issues in their implementation. The most common practice (88.65%) adopts a simple code pattern. These findings are help for future research on this topic.

Based on our findings, we develop IctApiFinder, which detects incompatible API usage issues in Android Apps based on inter-procedural data-flow analysis. IctApiFinder detects incompatible API usage issues on 361 Apps out of the 1,425 Apps we tested. It is sound and can effectively reduce the false positives of Android Lint by 82.1%.

In the future, we plan to automatically verify the bugs IctApiFinder detected. We also plan to give useful fixing suggestions to developers by mining equivalent APIs on different Android SDKs.

ACKNOWLEDGEMENT

This work is supported by the Innovation Research Group of National Natural Science Foundation of China (61521092 and 61672492), the National Key research and development program of China (2016YFB1000402 and 2017YFB0202002), the National Natural Science Foundation of China (U1736208), and Australia Research Council grants (DP170103956).

REFERENCES

[1] 2018. Retrieved April 26, 2018 from https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store

[2] 2018. Amaze File Manager. Retrieved April 26, 2018 from https://f-droid.org/en/packages/com.amaze.filemanager/

[3] 2018. Android Development tools. Retrieved April 26, 2018 from http://www.appbrain.com/stats/libraries/dev

[4] 2018. Android: Platform codenames, versions, and API levels. Retrieved April 26, 2018 from https://source.android.com/setup/start/build-numbers

[5] 2018. AOSP: Android Open Source Project. Retrieved April 26, 2018 from https://source.android.com/

[6] 2018. Fix EIC issues for batphone. Retrieved April 26, 2018 from https://github.com/servalproject/batphone/commits/development

[7] 2018. Fix minSDKVersion for PortKnocker. Retrieved April 26, 2018 from https://github.com/xargsgrep/PortKnocker/commit/master

[8] 2018. Free and Open Source Android App Repository. Retrieved April 26, 2018 from https://f-droid.org

[9] 2018. Gradle build tool. Retrieved April 26, 2018 from https://gradle.org

[10] 2018. IDC: Smartphone OS Market Share. Retrieved April 26, 2018 from https://www.idc.com/promo/smartphone-market-share/os

[11] 2018. Lint API Check. Retrieved April 26, 2018 from http://tools.android.com/recent/lintapicheck

[12] 2018. Port Knocker. Retrieved April 26, 2018 from https://f-droid.org/en/packages/com.xargsgrep.portknocker/

[13] 2018. Save For Offline. Retrieved April 26, 2018 from https://f-droid.org/en/packages/jonas.tool.saveForOffline/

[14] 2018. Serval Mesh. Retrieved April 26, 2018 from https://f-droid.org/en/packages/org.servalproject/

[15] 2018. Understanding the Android Support Library. Retrieved April 26, 2018 from http://martiancraft.com/blog/2015/06/android-support-library/#fn:3

[16] 2018. WeTest: Professional and Reliable One-stop Testing Service. Retrieved April 26, 2018 from http://wetest.qq.com/

[17] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc.

[18] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 468–471.

[19] Steven Arzt. 2017. *Static data flow analysis for android applications*. Ph.D. Dissertation. Technische Universität.

[20] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

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

[22] Eric Bodden. 2012. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. ACM, 3–8.

[23] Todd J Green, Molham Aref, and Grigoris Karvounarakis. 2012. Logicblox, platform and language: A tutorial. In *Datalog in Academia and industry*. Springer, 1–8.

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

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

[26] Liu Jie, Wu Diyu, and Jingling Xue. 2018. TDroid: Exposing App Switching Attacks in Android with Control Flow Specialization. In *Proceedings of the 33rd International Conference on Automated Software Engineering*.

[27] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 15–24.

[28] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 1–10.

[29] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *International Conference on Compiler Construction*. Springer, 153–169.

[30] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing inaccessible android apis: An empirical study. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 411–422.

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

[32] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

[33] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and Scalable Context-sensitive Pointer Analysis via Value Flow Graph. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2464157.2466483

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

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

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

[37] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan. 2014. A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEEE Software* 31, 2 (Mar 2014), 78–86. https://doi.org/10.1109/MS.2013.142

[38] Meiyappan Nagappan and Emad Shihab. 2016. Future trends in software engineering research for mobile apps. In *Software analysis, evolution, and reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 5. IEEE, 21–32.

[39] Je-Ho Park, Young Bom Park, and Hyung Kil Ham. 2013. Fragmentation problem in Android. In *Information Science and Applications (ICISA), 2013 International Conference on*. IEEE, 1–2.

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 49–61.

[41] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. 2012. Understanding reuse in the Android Market. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 113–122. https://doi.org/10.1109/ICPC.2012.6240477

[42] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for fast and easy program analysis. In *Datalog Reloaded*. Springer, 245–251.

[43] Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296

[44] M. D. Syer, B. Adams, Y. Zou, and A. E. Hassan. 2011. Exploring the Development of Micro-apps: A Case Study on the BlackBerry and Android Platforms. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. 55–64. https://doi.org/10.1109/SCAM.2011.25

[45] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[47] L. Wei, Y. Liu, and S. C. Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 226–237.

[48] Daoyuan Wu, Ximing Liu, Jiayun Xu, David Lo, and Debin Gao. 2017. Measuring the declared SDK versions and their consistency with API calls in Android apps. In *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 678–690.

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

[50] Zhang Yifei, Li Yue, Tian Tan, and Jingling Xue. 2018. Ripple: Refection analysis for Android Apps in incomplete information environments. *Software: Practice and Experience* 8, 1419–1437.

[51] Zhang Yifei, Sui Yulei, and Jingling Xue. 2018. Launch-Mode-Aware Context-Sensitive Activity Transition Analysis. In *Proceedings of the International Conference on Software Engineering*. 598–608.

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