Automatically Detecting Incompatible Android APIs

PEI LIU, Beihang University, China
YANJIE ZHAO, Monash University, Australia
MATTIA FAZZINI, University of Minnesota, USA
HAIPENG CAI, Washington State University, Pullman, USA
JOHN GRUNDY, Monash University, Australia
LI LI, Beihang University, China

Fragmentation is a serious problem in the Android ecosystem, which is mainly caused by the fast evolution of the system itself and the various system customizations. Many efforts have attempted to mitigate its impact via approaches to automatically pinpointing compatibility issues in Android apps. We conducted a literature review to identify all the currently available approaches to addressing this issue. Within the nine identified approaches, the four issue detection tools and one incompatible API harvesting tool could be successfully executed. We tried to reproduce them based on their original datasets, and then empirically compared those approaches against common datasets. Our experimental results show that existing tool capabilities are quite distinct with only a small overlap in the compatibility issues being identified. Moreover, these detection tools commonly detect compatibility issues via two separate steps including incompatible APIs gathering and compatibility issues (induced by the incorrect invocations of the identified incompatible APIs) determination. To help developers better identify compatibility issues in Android apps, we developed a new approach, *AndroMevol*, to systematically spot incompatible APIs as they play a crucial role in issue detection. *AndroMevol* was able to pinpoint 397,678 incompatible APIs against the full history of the official Android framework and 52 customized Android frameworks spanning five popular device manufacturers. Our approach could enhance the ability of the state-of-the-art detection tools by identifying many more incompatible APIs that may cause compatibility issues in Android apps and foster more advanced approaches to pinpointing all types of compatibility issues.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: fragmentation, compatibility issue, Android frameworks

1 INTRODUCTION

Fragmentation has been a severe problem for the Android ecosystem for years. "Fragmentation" refers to the fact that there is a massive number of Android devices manufactured by different companies running different Android operating system versions, including both official and customized ones. This introduces inconsistencies in that certain apps can only function properly on devices running specific Android versions with certain device features (i.e., the apps crash or don't work properly on different devices), leading to so-called *app compatibility issues*, short for *compatibility issues* in the paper, which include forward compatibility issues and backward

Authors' addresses: Pei Liu, Beihang University, China, Pei.Liu@monash.edu; Yanjie Zhao, Monash University, Australia, Yanjie.Zhao@monash.edu; Mattia Fazzini, University of Minnesota, USA, mfazzini@umn.edu; Haipeng Cai, Washington State University, Pullman, USA, haipeng.cai@wsu.edu; John Grundy, Monash University, Australia, John.Grundy@monash.edu; Li Li, Beihang University, China, lilicoding@ieee.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2023/9-ART $15.00
https://doi.org/10.1145/3624737

Just Accepted

compatibility issues [45]. The typical forward compatibility issue refers to API deletion in newer Android releases. An Android app will not function properly in newer Android releases due to the invocations of such removed APIs. On the contrary, the backward compatibility issue implies new APIs have been added in newer releases. An Android app will not function properly on older releases if its implementation utilizes such added APIs.

Compatibility issues have been considered one of the most severe problems in the rapidly and constantly evolving Android ecosystem. They can be induced by incorrect invocations of APIs, which we call API-related, or incorrect permission acquisition, non API-related [26]. On the one hand, they negatively impact the users' experience, as apps with compatibility issues may not be able to install on users' devices or may crash at runtime. On the other hand, they also increase the difficulties of developing apps. The vast number of device-Android version combinations create many technical complexities for developers and testers, which are non-trivial and yet expensive to resolve without a proper infrastructure in place.

To address these issues, we focus on API-related compatibility issues. There has been a significant amount of research in analyzing the compatibility issues induced by incorrect API invocations in Android apps. In the area of static analysis, researchers have proposed various automated approaches to pinpointing one of the most common compatibility issues: evolution-induced compatibility issues,. These refer to Android apps implementing and functioning properly on certain versions of official Android systems but not on others as of provision of certain Android APIs on these systems. For example, Li et al. [45] have designed and implemented a prototype tool called CiD that mines the evolution of the official Android framework codebase to locate evolution-induced incompatible Android APIs, i.e., new methods introduced in or existing methods being removed from the latest framework versions, and detects compatibility issues via static analysis approach based on these identified incompatible APIs. He et al. [32] also introduced a detection tool called IctApiFinder to detect evolution-induced compatibility issues. They first harvested incompatible APIs by extracting APIs from the released Android SDKs from version API level 4 to API level 27 and determining if there are Android APIs newly added or deleted, called incompatible APIs, in a newer released version, and then detected compatibility issues (induced by the incorrect invocations of these incompatible APIs) with their static analysis implementation. Wei et al. [72] have proposed a prototype tool called Pivot for characterizing device-specific incompatible APIs, e.g., APIs that are available for certain devices but not for others. Huang et al. [34] deeply explored the compatibility issues caused by the evolution of callback APIs, and proposed CIDER, which utilizes a graph-based model to detect API callback compatibility issues. We refer to different types of compatibility issues according to the incompatible APIs causing such issues, such as callback methods induced compatibility issues [34] and evolution-induced compatibility issues [32, 45], etc.

Although related work has attempted to tackle compatibility issues, it has not yet been entirely clear what the strengths and weaknesses of state-of-the-art tools are and to what extent they are able to identify different types of compatibility issues in real-world Android apps. Furthermore, it is also unknown to which extent we can reproduce experimental results from related work and how well each of the tools compares with each other in terms of detecting different kinds of compatibility issues. In this work, we formulate these concerns into three research questions that we aim to answer through empirical evidence and experimental results. Our three research questions are summarized as follows:

- **RQ1: What is the state-of-the-art tool performance in Android compatibility issues detection?** We answer this question through a systematic literature review, aiming to identify the primary studies relevant to statically detecting Android app compatibility issues. Our review identified nine primary publications that have proposed automated approaches for characterizing Android app compatibility issues. After careful analysis, we summarize five identified types of API-induced compatibility issues: evolution-induced (method), evolution-induced (field), device-specific (method), device-specific (field), and

Just Accepted

override/callback methods. Unfortunately, none of the existing approaches can tackle all five types of API-induced compatibility issues. The most recent, ACID [53], can only handle three out of the aforementioned five types of compatibility issues.

- **RQ2: Can we replicate the experimental results yielded by state-of-the-art tools targeting compatibility issue detection?**
  Replicability studies are regarded as an essential method to confirm the reliability of existing research (including both experiments and datasets). They are becoming an important focus in the software engineering community. To answer our second research question, we confirm the reliability of existing state-of-the-art compatibility issues detection tools by reproducing their experimental results with their original datasets. Our experimental results show that the majority of these study results can indeed be reproduced. The remaining small number of inconsistent results (yielded by IctApiFinder and FicFinder) are mainly caused by updates of the tools (such as dependency fixes) and apps (due to the unrecorded Github versions of the apps).
- **RQ3: How well do the tools in issue detection compare with each other?**
  To answer this question we apply selected state-of-the-art tools on two common sets of benchmark apps: (1) 65 apps used by the authors of selected tools and (2) 645 apps selected from the AndroidCompass dataset [56]. Our experimental results show that (1) compatibility issues detection approaches that achieve their purpose via systematically harvested incompatible API rules (such as CiD and IctApiFinder) can identify significantly more issues than those having their rules summarized manually, and (2) the intersection among the results reported by the selected tools is relatively small.

Our empirical study shows that typical compatibility issue detection involves two separate steps, including incompatible APIs gathering and compatibility issue determination (induced by such incompatible APIs), which are all achieved by the detection tools, CiD, IctApiFinder, CIDER, and FicFinder. To be more specific, compatibility issues in our paper refer to the invocations of incompatible APIs without guard checks for specific Android versions and/or devices. However, we empirically find that our community has not yet defined systematic approaches for identifying incompatible Android APIs, including evolution-induced APIs and device-specific ones, and many incompatible APIs have been overlooked. Even for the ones that can be detected, the intersection among the results given by the state-of-the-art tools, which mainly focus on detecting compatibility issues induced by the evolution of official Android APIs, is also relatively small [1]. Since incompatible APIs play a crucial role in compatibility issue detection, we go beyond the state-of-the-art approaches by presenting a new prototype tool for harvesting incompatible APIs. It is worth mentioning that the identified detection tools (CiD, IctApiFinder, CIDER, and FicFinder, except for Pivot) are all issue detection artifacts. They not only gather incompatible APIs but also pinpoint compatibility issues in Android apps. However, the approaches of harvesting incompatible APIs utilized by these issue detection tools and Pivot are not systematic or evolution-induced oriented. Our tool, named *AndroMevol*, advances the approach to gathering incompatible APIs by taking into account both the fast evolution of the official Android framework and the system customizations made by different device manufacturers. To evaluate the effectiveness of the approach, we further propose to answer two more research questions:

- **RQ4:** How well does *AndroMevol* perform in automatically identifying incompatible Android APIs?
  We selected five popular Android brands (i.e., Huawei, Xiaomi, Oneplus, OPPO, and Samsung) and evaluated if *AndroMevol* is capable of automatically generating a list of incompatible Android APIs, including methods and fields, for helping researchers and developers pinpoint potential compatibility issues in Android apps. Using the full history of the official Android framework and 52 customized Android frameworks extracted

[1] It is a rough comparison. We will discuss this better in threats to validity.

Just Accepted

from devices of popular brands. Our *AndroMevol* approach was able to report 397,678 incompatible APIs that do not exist in all the considered platform versions.

- **RQ5:** How is *AndroMevol* compared with state-of-the-art approaches targeting automated collection of incompatible APIs?
  To answer this research question, we compare our tool with other state-of-the-art approaches, CiD, CIDER, IctAPIFinder, FicFinder, and Pivot, in harvesting incompatible APIs. Compared with these selected tools, our *AndroMevol* approach can pinpoint many more incompatible APIs, enhancing the ability to detect compatibility issues in Android Apps for the state-of-the-art issue detection tools.

This work extends our conference paper [51] by proposing to the community a new prototype tool called *AndroMevol* to systematically harvest incompatible APIs. Our experimental results show that *AndroMevol* is effective in harvesting incompatible Android APIs. It also outperforms state-of-the-art approaches by harvesting at least eight times more incompatible APIs, including 195,883 APIs that have never been reported previously. Furthermore, we have updated the Discussion and Related Work sections to cope with the aforementioned changes. The source code and datasets are all made publicly available in our artifact package via the following link:

https://github.com/MobileSE/AndroMevol

In summary, the main contributions of this paper are:

- A systematic literature review across 19 top-tier venues to apprehend the status quo of compatibility issue detection. We carefully read the targeted nine relevant papers from the selected top venues, summarized five types of compatibility issues, and identified seven state-of-the-art detection tools to pinpoint such issues.
- A comprehensive comparative study to compare the ability of each identified detection tool. As the detection tool ACID and ACRYL cannot be replicated and Pivot is developed only for incompatible APIs collection, we could successfully replicate four detection tools, including CiD, IctApiFinder, CIDER, and FicFinder. To measure the ability of the issue detection, we run the detection tools against two datasets. One is the original dataset published along with the detection tools and the other is the dataset AndroidCompass, which contains open-source Android projects collected from GitHub. Our evaluations on the original dataset demonstrated that CiD outperforms the other detection tools by detecting more than one and ten times the number of compatibility issues pinpointed by IctApiFinder and FicFinder, respectively. We also found that typical issue detection approaches involve two separate steps, including incompatible APIs identification and compatibility issue determination, and incompatible APIs play a crucial role in issue detection. To boost the ability of the issue detection of the tools, a complete set of incompatible APIs is necessary.
- A systematic approach to harvesting incompatible APIs, including methods and fields, among five popular OS brands, including Huawei, Xiaomi, Oneplus, OPPO, and Samsung alongside AOSP. We proposed the new prototype tool called *AndroMevol* to harvest incompatible APIs from the official Android frameworks and 52 customized Android frameworks extracted from the released ROMs of the five popular brands. With the *AndroMevol*, we could collect a total of 397,678 incompatible APIs including 195,883 previously untouched ones, which could be accessed by Android apps inducing compatibility issues. Compared to other API harvest tools, *AndroMevol* performs better by harvesting at least eight times more incompatible APIs.

2 STATE-OF-THE-ART APP COMPATIBILITY DETECTION APPROACHES (RQ1)

We performed a systematic literature review to understand the current state-of-the-art in Android app compatibility analysis approaches and available tools and datasets.

Just Accepted

2.1 Literature Review

Figure 1 illustrates the working processes of our literature review, summarized based on the guidelines provided by Kitchenham et al. [36] and Brereton et al. [24], as well as lessons learned from our own recent SLR practices [38, 52, 63, 78].

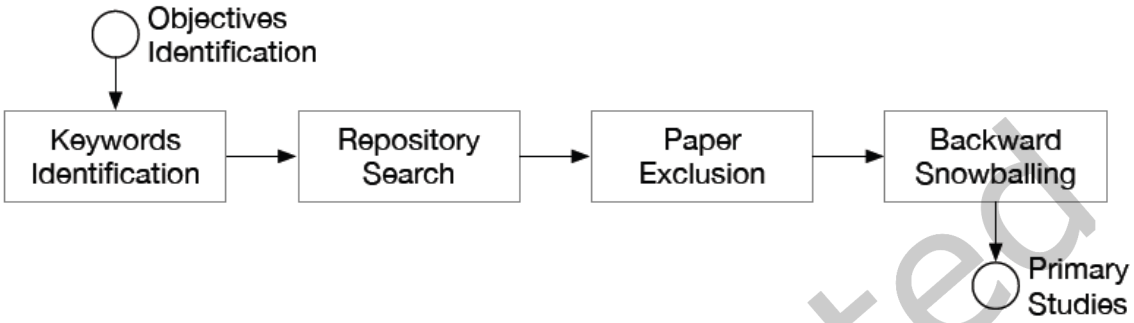

Fig. 1. Overview of the literature review process.

Keyword Identification. To understand the status quo of incompatible app analyses, we use a set of keywords to search for key relevant publications in popular repositories. The keywords we leveraged are essentially made up of two groups (i.e., G1 and G2). Each group contains several keywords. The search string is then formed as a combination, i.e., g1 AND g2, where g1 and g2 are formed each as a disjunction of the keywords respectively from groups G1 and G2.

$G1 : android, mobile, *phone*$

$G2 : *compati*, deprecat*, issue*, evolution$

Repository Search. To focus our search, we applied these keywords on all the CORE[2] A/A* ranked venues. This keeps the review process lightweight while ensuring that important related works are not missed. In the software engineering field (i.e., containing 'software' keyword in the venue title and falling in the following fields of research code: 0803 for journals and 4612 for conferences), as summarized in Table 1, there are 19 venues (5 journals and 14 conferences) ranked as A/A* by CORE. These are all the top SE publication venues where high quality Android app compatibility detection work is typically published. We then went through these 19 venues one by one and applied the aforementioned keywords to search for relevant publications. Eventually, we were able to locate 44 publications across 13 venues (i.e., there is no relevant paper identified in 6 of the venues).

Table 1. CORE A/A* ranked software engineering venues.

Type	Source	Venues
Journals	CORE2020	TOSEM, TSE, EMSE, JSS, IST
Conferences	CORE2021	ASE, ESEC/FSE, ICSE, EASE, ECSA, ISSRE, ESEM, ICSME, MSR, ICSA, SANER, SEAMS, ICST, ISSTA

Paper Exclusion. As we aimed at collecting as many relevant papers as possible, we have simply considered all the returned results. However, not every paper is related to automated Android app compatibility issue

[2]https://www.core.edu.au/home

Just Accepted

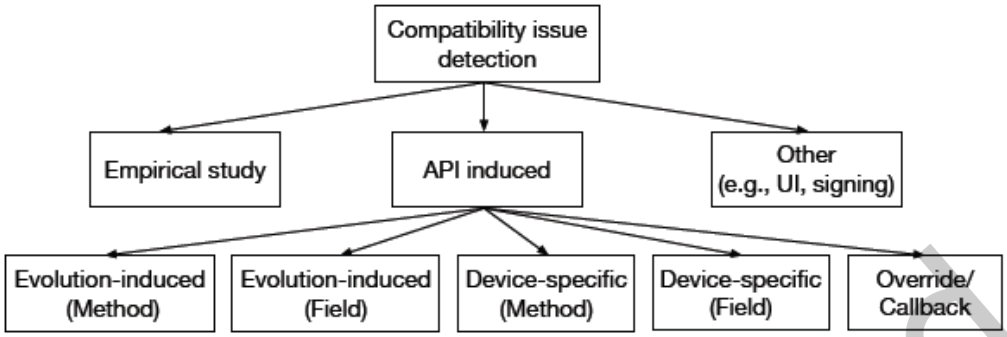

Fig. 2. The category of the papers targeting compatibility issues on the Android platform.

detection. We there go one step further to read the abstract (and full content if needed) of the obtained papers to only retain the closely related ones by applying the following exclusion criteria: (1) Short papers (i.e., less than six pages in double-column format or 11 pages in single-column format) are excluded. (2) Papers targeting non-Android mobile devices are excluded. (3) Papers targeting Android but that do not concern compatibility issues are excluded. (4) Papers targeting Android compatibility issues but that do not concern API-induced ones are excluded (categorized as Other in Figure 2). For example, the work presented by Ki et al. [37], which proposes an automated testing framework for Android apps named Mimic for characterizing UI compatibility issues, is excluded. Another work presented by Wang et al. [68], which has discussed a type of app signing compatibility issue introduced by unsupported digest/signature algorithms, is also excluded. (5) Papers targeting Android compatibility issues but that do not introduce automated approaches to detect or resolve them are excluded (categorized as Empricial Study in Figure 2). For example, Nielebock et al. [56] contribute an Android compatibility check dataset named AndroidCompass, which comprises changes to compatibility checks in the version histories of the Android projects. Cai et al. [26] conduct a large-scale study of compatibility issues based on Android apps developed over the past eight years to comprehend the symptoms and root causes. These papers do not introduce a prototype tool to detect compatibility issues in Android apps and hence are excluded. After applying these exclusion criteria, there are nine papers retained that are closely related to automated incompatible Android API detection.

Backward Snowballing. Based on the papers identified in the previous steps, we conducted a backward snowballing approach to ensure that important closely related papers (e.g., with titles not matching our search string or published outside of the selected 19 venues) are not missed by our lightweight literature review. For each paper we carefully read the related work part and attempted to find cited papers that are closely related to our study but have not yet been included. This process did not help us identify any new papers, suggesting that the keywords and venues that we selected to search for relevant publications are indeed the most relevant ones.

2.2 Result

In total, our Systematic Literature Review (SLR) search process identified nine relevant papers (hereinafter referred to as primary studies, which are listed in the first column of Table 2. The nine papers are collected from seven venues with publication dates ranging from 2016 to 2021 (cf. second and third columns in Table 2). The last column describes the availability of these tools. Some of them are open-sourced, while others are published as executable files on the associated papers' websites.

Just Accepted

Table 2. Full List of Collected and Examined Papers.

Tool/Reference	Year	Venue	Tool availability
ACID[53]	2021	SANER	Available [1]
ACRYL (extension)[62]	2020	EMSE	Open Source [2]
ACRYL[61]	2019	MSR	Open Source [2]
Pivot[72]	2019	ICSE	Available [5]
CiD[45]	2018	ISSTA	Open Source [3]
IctApiFinder[32]	2018	ASE	Open Source [7]
CIDER[34]	2018	ASE	Available [4]
FicFinder (extension)[73]	2018	TSE	Available [6]
FicFinder[71]	2016	ASE	Available [6]

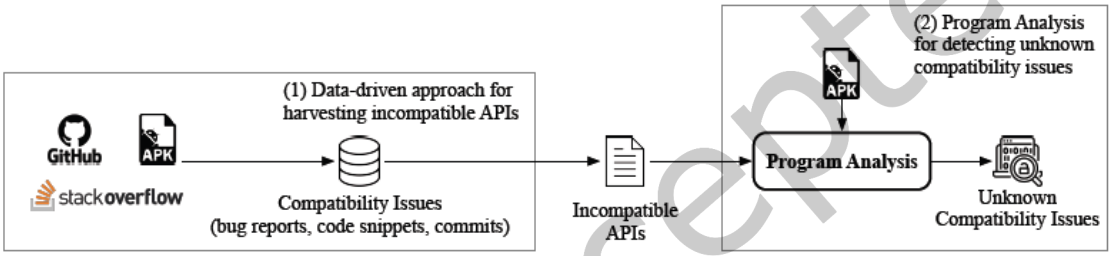

Fig. 3. The typical working process of detecting Android compatibility issues.

Just Accepted

2.3 State-of-the-Art App Compatibility Analysis Approaches

After identifying the primary publications, we carefully read their full papers to understand how each of their automated compatibility issues detection approaches is implemented. We then summarize the common working process taken by those approaches to detect Android compatibility issues.

As shown in Figure 3, the objective is often achieved via two steps: (1) data-driven approach for harvesting incompatible APIs and (2) program analysis for detecting unknown compatibility issues. The output of the first step will be a list of incompatible APIs, which will be taken as input to the second step. With the two typical steps of the working process of compatibility issue detection, we summarized the collected tools as in Table 3. The second and third columns describe incompatible APIs collection and issue detection per se separately. CIDER and FicFinder only support issue detection while Pivot only focuses on incompatible APIs collection regardless of publicly available ones or conventionally restricted ones. The remaining tools are working as a whole supporting both APIs harvesting and issue detection.

After carefully reading their full content, we understand that compatibility issues stem from API inconsistencies that are induced by the evolution of Android OS per se and different OS customizations. We, therefore, categorize compatibility issues into five types as described in Figure 2 according to the different types of underlying inconsistent APIs. For each of the considered tools, we further summarize and list its capabilities in Table 4. It is worth noting that FicFinder does have the ability to detect device-specific methods-induced compatibility issues but the incompatible APIs leveraged by the artifact are all evolution-induced ones. Therefore, we assume the artifact FicFinder could only detect evolution-induced compatibility issues in the following discussion. Columns

Table 3. Working Process Support of Tools.

Tool/Reference	API Harvest	Issue Detection
ACID[53]	✓	✓
ACRYL (extension)[62]	✓	✓
ACRYL[61]	✓	✓
Pivot[72]	✓	✗
CiD[45]	✓	✓
IctApiFinder[32]	✓	✓
CIDER[34]	✗	✓
FicFinder (extension)[73]	✗	✓
FicFinder[71]	✗	✓

2-6 describe the detection of the five types of compatibility issues, which are further detailed with concrete examples as follows.

Table 4. Examination results of the approaches proposed in the retained primary studies.

Tool/Reference	Evolution-induced (Method)	Evolution-induced (Field)	Device-specific (Method)	Device-specific (Field)	Override/ Callback	Systematic (Sound)	Fully automatic[2]
ACID[53]	✓	✓[1]	✗	✗	✓	✓	✓
ACRYL (extension)[62]	✓	✗	✗	✗	✗	✗	✓
ACRYL[61]	✓	✗	✗	✗	✗	✗	✓
Pivot[72]	✓	✗	✓	✗	✗	✗	✓
CiD[45]	✓	✗	✗	✗	✗	✓	✓
IctApiFinder[32]	✓	✓[1]	✗	✗	✗	✓	✓
CIDER[34]	✗	✗	✗	✗	✓	✗	✗
FicFinder (extension)[73]	✓	✗	✓	✗	✗	✗	✗
FicFinder[71]	✓	✗	✓	✗	✗	✗	✗

[1] Only mentioned but not illustrated in detail.
[2] There is no human involvement in the core process, e.g., the learning/knowledge collection phase.

Evolution-induced (Method): The signatures [3] of some public methods are altered (i.e., removed, newly added, or parameter type changes, etc.) during the evolution of the framework. Figure 4a demonstrates such an example, for which the code snippet is initially reported in [32], where statements beginning with the + signs indicate a possible fix for this incompatibility. The API *startDrag()* called on Line 7 is introduced into SDK after level 11. However, the minSdkVersion of this app is set to 10. Consequently, if not protected with the "if-else" block, a "NoSuchMethodError" exception will be thrown, leading to crashes on devices running SDK version 10.

Evolution-induced (Field): During the evolution of the framework, the signatures of some publicly accessible fields could also be altered (i.e., removed or newly added). Unfortunately, apart from [32] and [53], none of the other papers discusses such issues. Moreover, no relevant examples are given in all the research papers. Then we use an example that we discovered throughout our research. There is an evolution-induced issue with a field called "BitmapFactory.Options.inDither" at Line 4 of Figure 4b. It's supported by API Levels 1 through 23, however, since API Level 24, it's been deprecated, creating compatibility issues when an app sets a target SDK version equal to or greater than 24.

Device-specific (Method): Due to the customization of smartphone manufacturers, some APIs only work on some devices but not on others. Figure 4c demonstrates such an example, originally reported by Wei et al. [72].

[3]The method signature in our research and current related work refers to the combinations of the method return type, method name, and method parameters type list.

Just Accepted

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

(a) Example 1: Evolution-induced(Method)

```
1  public static Bitmap getCachedArt(final Context context,
               final Song song){
2    ...
3    Options options=new Options();
4    options.inDither=false;
5    options.inPreferredConfig=ARGB_8888;
6    ...
7  }
```

(b) Example 2: Evolution-induced(Field)

```
1  Camera mCamera = Camera.open();
2  Camera.Parameters params =  mCamera.getParameters();
3  ......
4  + if (android.os.Build.MODEL.equals("Nexus 4") {
5  +  params.setRecordingHint(true);
6  + }
7  ......
8  mCamera.setParameters(params);
9  mCamera.startPreview();
```

(c) Example 3: Device-specific(Method)

```
1  private static HttpClient getNewHttpClient() {
2    ...
3    sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
4    ...
5  }
```

(d) Example 4: Device-specific(Field)

```
1   public void onAttach(Context context) {
2    super.onAttach(context);
3   - mActivity = (BrowserActivity) context;
4   - ......
5   + attachActivity((BrowserActivity) context);
6   }
7   + public void onAttach(Activity activity) {
8   +  super.onAttach(activity);
9   +  if (Build.VERSION.SDK_INT < 23) {
10  +    attachActivity((BrowserActivity) activity);
11  +  }
12  + }
13  + private void attachActivity(BrowserActivity activity) {
14  +  mActivity = activity;
15  +  ......
16  + }
```

(e) Example 5: Override/Callback

Fig. 4. Code Examples

Only if the result of the conditional statement for checking the device identifier according to "Nexus" is true, that is, the corresponding app is indeed running on "Nexus", the API *setRecordingHint()* on Line 5 will be executed.

Device-specific (Field): Similar to evolution-induced compatibility issues, the customization of Android frameworks can also introduce incompatible fields (i.e., exist in some devices but not in others), referred to as device-specific fields. No code example is provided in our reviewed primary papers, similar to evolution-induced (field). We then take the example of "<org.apache.http.conn.ssl.SSL-SocketFactory: org.apache.http.conn.ssl.X509HostnameVerifier ALLOW_ALL_HOSTNAME_VERIFIER>", as shown in Figure 4d. According to our analysis results, this field is not supported by OPPO smartphones in the SDK of API Level 26, which account for more than 10% of global smartphone shipments [8]. If an app that uses this field is installed and run on an OPPO smartphone with SDK version 26, compatibility issues may arise.

Override/Callback: Due to the evolution of the Android framework, some callbacks may have been altered. Here, the callbacks are methods defined by the framework that could be explicitly overridden[4] by client Android developers, and their execution will be triggered by the framework. Figure 4e demonstrates such an example excerpted from [34]. The *onAttach(Context)* callback method at Line 1 is introduced from API level 23. This callback method will not be executed if this code is run on a smartphone with an API level lower than 23. Thus it could cause the *mActivity* field not to be initialized, and a "NullPointerException" may be thrown when using it.

The table shows that most of the tools are developed for detecting compatibility issues induced by the method evolution of the Android system. For the field evolution-induced compatibility issues, ACID and IctApiFinder have mentioned the issue in the corresponding papers but did not explain the issue in detail. Pivot and FicFinder

[4]Actually, all the methods that are declared as public or protected could eventually be explicitly overridden by client apps. In this work, we take all of such methods into account and hence will not differentiate (and hence specifically emphasize) if the given method is a callback.

Just Accepted

also considered compatibility issues induced by methods provided by specific devices, while none of the detection tools examined compatibility issues resulted from fields carried by specific devices. For the independent issue induced by the evolution of callback methods, CIDER is the only approach developed intentionally to handle this, while ACID considered both evolution-induced and this special one. To summarize, unfortunately, none of these approaches have considered all the identified types of compatibility issues. The most recent approach, ACID [53], can only handle three out of the aforementioned five types, leaving device-specific issues unaddressed. It is also worth noting that the two approaches, which have indeed taken evolution-based fields into account, have only mentioned this capability but do not elaborate further with the support of experimental evidence.

Furthermore, in column 7 of Table 4, we further summarize whether the proposed approach involves a systematic approach to harvest an incompatible API list (hence the results can be considered complete). As summarized in Table 4, only three approaches (i.e., ACID, CiD, IctApiFinder) leverage a systematic approach to harvest incompatible APIs. The majority of considered approaches only take ad-hoc approaches aiming at detecting as many compatibility issues as possible without endeavoring to identify all the possible compatibility issues, i.e., the compatibility issues are not discovered following a systematic approach aiming at covering all the possible cases. As an example, Scalabrino et al. [61] present an automated compatibility issue detection approach called ACRYL, which leverages the knowledge collected from changes implemented in other apps responding to API changes to achieve its purpose. Such an approach, although implemented in an automated manner, cannot collect all the possible compatibility issues lying in the Android ecosystem and thereby can unfortunately yield false-negative results.

Finally, the last column further highlights whether the proposed approach itself is fully automated or not. An automated approach should not involve any manual efforts that may pose difficulties to replicate. Among the selected nine approaches, six of them do provide automated ways to identify compatibility issues (i.e., misuse of incompatible APIs) in real-world Android apps. Three approaches rely on manual efforts to achieve their objectives, making them not extensible (at least in an easy way) to detect newly introduced compatibility issues. For example, Wei et al. [71, 73] have empirically studied the fragmentation-induced issues to portray the symptoms and root causes of compatibility issues and subsequently proposed a static-analysis tool named FicFinder to detect such compatibility issues. The major limitation of FicFinder is the requirement of manual efforts to build the patterns of API/context pairs, which are summarized from the aforementioned empirical study. Such manual efforts are expensive to be extended to summarize more compatibility issues.

RQ1 Findings

In our analysis, no state-of-the-art approaches are capable of detecting all five types of compatibility issues that have been identified to date, and some of them require considerable manual effort.

3 REPLICABILITY STUDY (RQ2)

The second research question aims at checking to what extent can we replicate the experimental results yielded by the state-of-the-art tools targeting compatibility issues detection.

3.1 Tool Selection

Ideally, we would like to consider all the tools to perform the replicability study. Among the nine primary studies, there are, in total, seven tools worth reproducing. ACRYL and FicFinder have respectively been first presented in a conference paper and then extended to a journal paper. In these two cases, only the tool versions presented in the latest paper are considered. Among the seven tools, we decide to exclude Pivot as it does not really involve the actual detection of compatibility issues in Android apps, as highlighted in Table 3. For the remaining six detection tools, we download all of these different tools from their published site and contact the authors of

Just Accepted

the tools to make sure if the tools *per se* and the experimental datasets are the same as they were presented in the original papers. The developers confirm that IctApiFinder [32] has been updated due to the evolution of dependencies. We then try to execute them one-by-one in our local environment to make sure they can be successfully reproduced. Unfortunately, we have to further exclude ACID and ACRYL from consideration as these two tools cannot be successfully executed. We have contacted their authors for clarification, but until now, we still cannot properly execute them. Therefore, we conduct the reproducibility study based on the remaining four tools, which are detailed as follows.

CiD [45] first models the lifecycle of publicly available Android APIs by extracting Android APIs from Android framework source code and then analyses Android Apps including both the primary app code and extra code. However, it is uncertain whether the Android app has accessed a problematic Android API or not just by checking if the app contains an invocation of the problematic Android API as the problematic Android API can also be protected by SDK version checkers. Therefore, the authors proposed a path-sensitive inter-procedural backward data-flow analysis to verify if the problematic Android APIs are protected with API-level related conditions. A compatibility issue is identified once the API is not protected by version condition checks and the API is not supported in the range designated in AndroidManifest.xml.

IctApiFinder [32] first conducts an extensive empirical study over 11 consecutive Android versions and approximately 5,000 Android Apps. The authors find that many different APIs are released between two consecutive Android API releases and thus App developers or third-party library developers provide additional code to guarantee the same behaviours on different OS versions. More importantly, the additional supporting code shares the same pattern that is SDK version check. With the provided SDK version check, different Android APIs are invoked to run smoothly on different OS versions. Based on these findings, they propose the tool by first building the interprocedural control flow graph (ICFG) by Soot for Android Apps and then extracting Android APIs (including the publicly available Android APIs and the restricted APIs with the access modifier protected) from SDK (android.jar) file as the authors believe that it is not accurate to extract from the SDK document api-version.xml. With the ICFG, it transfers the dataflow analysis problem into a reachability problem. For each Android API in the ICFG, the tool detects if it is supported in the defined API levels interval in AndroidManifest.xml as there are different SDK version constraints (conditional SDK version check to access the Android APIs) in different program points. If the designated API levels are not supported at a certain point, an issue is detected.

CIDER [34] focuses on compatibility issues caused by callback APIs as the evolution of Android frameworks. With the help of an empirical study, they find that two common types of callback API evolutions: API reachability change and API behavior modification can change app control flows and induce compatibility issues. Thus, they leverage the concept of Callback Control Flow Graph (CCFG) [77] and propose a graph-based model, Callback Invocation Protocol Inconsistency Graph (PI-Graph), to capture the structural invocation protocol inconsistencies to detect callback-induced compatibility issues (inconsistent app control flows) when apps running on different API levels. The authors first encode seven different PI-Graphs related to 24 key Android APIs from their empirical dataset and then implement the detection tool based on Soot [67].

FicFinder [73] is actually the first seminal work to better understand fragmentation-induced compatibility issues and detect these issues via the proposed approach. By conducting empirical study and investigating real-world compatibility issues, the authors found that the majority of the issues are induced by the improper use of Android APIs in a problematic running environment, which is called issue-triggering context and the context can be expressed in context-free grammar. Therefore, the algorithm identifies the issue-inducing Android APIs as well as their dependencies, analyses the calling context, and then compares it with the modeled issue-triggering context. To analyze the dependencies issue-inducing API related, the algorithm carries out an inter-procedural backward slicing on the call site to acquire the slices of statements on the basis of program dependence graph [29]. If the triggering context is not considered before invoking the API, a new issue is reported. To implement this artifact, the well-known static analysis framework, Soot [67], is utilized.

Just Accepted

Each of the selected tools requires a specific configuration. As the detection result relies on these basic configuration parameters, we investigated the tool document and configuration setup process and tried to align the configurations between these selected tools to make sure they do have a similar configuration.

3.2 Datasets

Recall that, with RQ2, we are interested in evaluating the replicability of the selected tools. We aim to achieve this by running the tools against their original datasets. We, therefore, request the tools' authors to share their datasets, including mainly the ones with results manually confirmed by the authors[5] and have been explicitly discussed in their manuscripts (hence can be compared). To this end, we have eventually selected 65 apps, which are made up of (1) twenty Android apps for CIDER, seven apps for CiD, eight apps for IctApiFinder, and thirty for FicFinder.[6]It is worth reminding the readers that we have to exclude some of the shared apps because they are no longer available on the web and hence the apps cannot be downloaded based on the information shared by the authors, or the shared source code snippets cannot be compiled to Android apps. Nevertheless, this exclusion of a small number of apps including 12 from IctAPIFinder and 23 from FicFinder (12 + 23 = 35) should not impact the results of the replicability study.

3.3 Results

When carried out our replication, CIDER and CiD were found to have exactly the same outputs on the original Android Apps. In contrast, FicFinder and IctApiFinder have some different outputs with regard to their original experimental Apps. We now detail their differences respectively.

IctApiFinder. The artifact was developed along with the paper in 2018 and was not open-sourced till 2021. With the acquired eight exact Android Apps, we can successfully run the tool on all of them. However, 6 of them do have a different number of issues reported compared with the original paper. Among the six different apps, the paper in total reported 49 issues regardless of TP (True Positive) and FP (False Positive), while our experiment reveals 108 compatibility issues. As we cannot obtain the original results rather than the reported number of issues, we cannot know which issues are different compared with the original results. One reason explaining the differences could be that, as also confirmed by the authors, the tool has not been maintained during the last three years. Therefore, there are some dependencies that are not available anymore, and also, there are some APIs not supported in the newer updated dependencies. To release the project, the authors replaced it with newer versions of dependencies and commented out some non-supported APIs in the project. The authors further noted that they could not make sure if such updates have bad or good effects on the final detection results.

FicFinder. The artifact was first published in 2016 and then extended in 2018. We can successfully execute the artifact on all of the Android projects. The paper describes the detected results in two different categories. One is compatibility issues in TP and FP, and the other is Good Practice (GP) meaning already fixed issues. After we reproduce the artifact with the original experimental Android app dataset in our local environment, seven of them have different outputs compared with the original ones presented in the original published paper. Among the seven apps, we find that 2 of them have the same total number of detected results but have a different number of compatibility issues and good practices, such as GadgeBridge [10] was reported one detected issue (regardless of TP and FP) and one GP but we reproduced with 2 issues detected, AnkiDroid [9] was reported 4 GP detected but we reproduced with 4 issues. The remaining five apps further have a different total number of detected results, such as LibreTorrent [11] revealed 6 GP but we detected only 3 GP, MozStambler [13] contained 1 issue and one

[5]We decide to not request the full dataset leveraged by the authors because it may involve a very large number of apps that are not convenient to share.

[6]The FicFinder authors have actually considered 53 Android projects but only thirty of them can be compiled into Android APKs. Although FicFinder can take either Android APKs or disassembled class files as input, we will only replicate the capability of analyzing Android APKs, which are also the input of the other considered tools.

Just Accepted

GP but we only detected 1 issue. The possible reason behind this is that they did some regular updates on the artifact as the authors still utilize this one in their research, such as the case study in their newer work Pivot.

To summarize, as revealed by our study, most of the experimental results yielded by the selected four tools could be reproduced. The small number of cases that cannot be reproduced are mainly due to tools' updates, either because of lacking maintenance so that we have to arbitrarily update some dependencies to make it runnable in practice or intentional evolutions to keep improving its capabilities. Such updates, either intended or not, have indeed caused difficulties in reproducing the exact original results. Therefore, we argue that there is a need to always record the artifacts, along with the experimental datasets such as Android apps including both source code and bytecode APK files if possible, in permanent sites (e.g., Zenodo or Figshare). The artifacts should also be well-configured in docker-alike containers that can support direct execution of the tools and hence mitigate unnecessary dependency errors that may hinder the tools' replicability.

RQ2 Findings

Most of the experimental results yielded by the four selected state-of-the-art tools can indeed be reproduced. There are, however, a small number of non-replicated cases that are mainly caused by slight updates of the tools or the evaluated apps.

4 COMPARISON STUDY FOR ISSUE DETECTION (RQ3)

This research question aims to empirically compare the state-of-the-art tools targeting the detection of compatibility issues in Android apps. We rely on the validity provided by the artifacts themselves. We answer this research question by first presenting the experimental setup (including tool selection and datasets) in Subsections 4.1 and 4.2 and then the experimental results in Subsection 4.3.

4.1 Tools Selection

Recall that there are only four tools that we can replicate to scan compatibility issues (as discussed in the previous section). Therefore, we select the same four tools to achieve this objective in this work, i.e., comparing these four tools w.r.t. their compatibility issues detection capabilities.

4.2 Datasets

In this work, we use the following two datasets to support our comparison study in compatibility issues detection.

- **Dataset1**: The same 65 apps used for the replicability study as discussed in Section 3.
- **Dataset2**: 645 Android apps selected from the total 1,375 open-source apps of AndroidCompass dataset [56]. AndroidCompass contains a dataset of git commits related to Android compatibility checks (including evolution-induced, device-specific, and override/callback-related ones), which are originally harvested from 1,375 open-source Android projects on Github. Some git commits contain compatibility issue fixes (e.g., adding compatibility checks for APIs that are not protected initially), while others do not (e.g., adding new Java files that include compatibility checks). In this work, we are only interested in the former ones as based on which we could locate problematic app versions containing actual compatibility issues. This study will leverage this information as partial ground truth to support the comparison study. We collect Android projects with git commits containing API version guard checks (i.e., checking the Android versions just before the API invocations). Going one step further, we reset the commit of the selected projects just before the fix commit (i.e., adding Android version checks) and compiled them into installable APKs. Unfortunately, several app projects are no longer available on Github, while some others cannot be easily

Just Accepted

Table 5. Experimental results obtained based on the 65 apps located in Dataset1.

App Name	Callback-induced	Evolution-Induced		
	CIDER	FicFinder	IctApiFinder	CiD
Tinfoil-Facebook	0	1	1	2
kolabnotes	1	4	5	6
SteamGifts-chocolate-debug	0	6	20	23
OsmAnd	1	3	7	44
iFixitAndroid	0	7	58	218
Simple-Solitaire	1	0	1	10
Anki-Android	0	6	150	86
login-sample-debug	4	0	2	3
ooniprobe-android-1.3.1-debug	1	0	4	38
APICompatibility_Inheritance	0	0	2	3
APICompatibility_Varargs	0	0	2	2
SurvivalManual-4.1-debug	0	2	1	15
Calendula	0	15	29	63
libretorrent	3	1	13	59
APICompatibility_Protection2	0	1	1	0
StreetComplete	0	2	7	5
red-moon	0	0	13	21
padland	1	0	13	4
duckduckgo-0.6.0-release	1	0	1	2
transdroid	0	1	214	37
materialistic-hacker-news	0	1	32	36
materialfbook	0	1	15	38
ownCloud	0	2	66	181
AndStatus	0	2	43	27
RedReader	0	1	30	7
opentasks-1.1.8.2	0	24	14	51
APICompatibility_Basic	0	0	1	1
Gadgetbridge	0	2	21	35
Total	13	82	766	1,017

Just Accepted

compiled into APKs (e.g., due to outdated dependencies), so we have to exclude them. Eventually, we were able to collect 645 apps (at least one compatibility issue in each of them) to build this dataset.

4.3 Results

Results on Dataset1. We first apply the selected tools to analyze the apps in Dataset1. Unfortunately, 37 apps cannot be handled successfully by both IctApiFinder and CiD (i.e., 24 and 15 failures, respectively). The

corresponding error messages, such as worker thread execution failed [7], Dex version is not supported [8], and IllegalArugmentException [9], etc., indicate that the failures are mainly raised by Soot, the underlying static analysis framework leveraged by these two tools. This problem has been discussed by the authors in their article as a potential threat to validity. It is also a well-known problem when performing static analysis on top of Soot.

For the remaining 28 successfully analyzed apps, Table 5 presents the detection results. CIDER, different from the other three detection tools, was developed for callback-induced compatibility issues. Among the 28 apps, only 8 apps are reported to include callback-induced issues. The reason behind this small number could be explained by the fact that the tool only leverages seven manually summarized rules to detect such issues. Such a manual process may not be able to include all the different situations and hence may lead to incomplete results. Similarly, FicFinder, which leverages 20 manually summarized incompatible APIs, reports only 82 compatibility issues, which are also significantly fewer results compared with the remaining two tools that have leveraged systematic approaches to harvest incompatible APIs (as indicated in Table 3). The typical working process for these selected issue detection tools as shown in Figure 3 in the paper includes two steps. The first one is incompatible APIs gathering and the second step is issue detection via program analysis by incorporating the incompatible APIs harvested in the first step. The number of collected incompatible APIs is strongly correlated with the final detected number of compatibility issues. The number of incompatible APIs leveraged by CiD and IctAPIFinder is much more than others. The more incompatible APIs imply more potential incompatibility issues. This experimental result further confirms that it is essential to invent systematic approaches to harvest incompatible APIs so as to support automated compatibility issues detection in Android apps.

Both IctApiFinder and CiD yield significantly more results than FicFinder (revealed by the total number of detected issues in Table 5 classified into two types, including callback-induced and evolution-induced, according to the targets of the selected detection tools) since they do take systematic approaches to collect much more incompatible APIs compared with the limited manually harvested ones in FicFinder, which demonstrates the importance of the incompatible APIs gathering for the issue detection. However, the detection results of IctApiFinder and CiD are quite different. Among the 28 apps successfully analyzed by both of these two tools, IctApiFinder and CiD respectively yield a total of 766 and 1,017 issues, for which only 52 were reported by both of them. This experimental result is quite surprising as we would have expected that IctApiFinder and CiD would have much more overlap in terms of their detected compatibility issues. We, therefore, go one step deeper to investigate why these two tools yield quite different results, i.e., being able to locate a quite number of compatibility issues while also missing many of them reported by the other tool. We look at the number of distinct incompatible APIs detected by these two tools. Our analysis shows that the total 766 and 1,017 compatibility issues reported by IctApiFinder and CiD are essentially caused by 147 and 551 incompatible APIs, respectively. As highlighted in Figure 5, the intersection between these two incompatible APIs sets is quite small (i.e., only 63 out of 551 incompatible APIs considered by CiD are also taken into account by IctApiFinder). One reason causing this difference is that different framework versions are considered (e.g., the incompatible APIs collected by IctApiFinder are from 4 to 27, while CiD is from API 1 to 25). Subsequently, the common compatibility issues reported by both of these two tools will be small as well. We could not configure the same incompatible APIs from the same range of API levels as different detection tools require different inputs. For example, FicFinder requires JSON-like configuration of incompatible APIs as input and IctApiFinder takes additional third-party tools, such as Heros [23], Doop [64], LogicBlock [20, 31], etc., to extract incompatible APIs. Even though a fair comparison is not possible, we could still have such a comparison to have a basic understanding of the ability of issue detection among the collected detection tools. As is known to us, the attribute `minSDKVersion` indicating

[7] https://github.com/soot-oss/soot/issues/1279
[8] https://stackoverflow.com/questions/49606951/dexexception-not-support-version
[9] https://github.com/soot-oss/soot/issues/331

Just Accepted

the minimum API level on which the apps could be run is always greater than API level 4 among nowadays available apps. Among the dataset of a total of 710 (65 + 645) Android apps in our experiments, only 4 of them set the `minSDKVersion` to 4, only accounting for 0.56% (4/710). The lower API level of the API range does have little impact on the issue detection results.

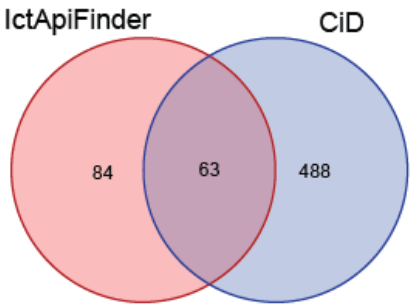

Fig. 5. Venn diagram of incompatible APIs utilized in IctApiFinder and CiD.

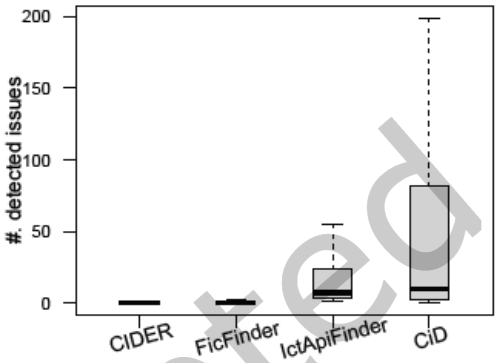

Fig. 6. Compatibility issues detected by different detection tools against Dataset2.

Results on Dataset2.

We then apply the selected tools on Dataset2, which contains a large number of real-world Android apps selected from the AndroidCompass compatibility checks dataset. Unfortunately, over half the apps are excluded from the dataset as they cannot be successfully analyzed by all the selected tools. Among the 277 remaining apps, CIDER, FicFinder, IctApiFinder, and CID have reported 12, 277, 5,009, and 27,874 compatibility issues, respectively. Figure 6 further illustrates the distribution of detected compatibility issues in real-world Android Apps. Clearly, CiD yields more issues than the other tools, followed by IctApiFinder and then FicFinder. CIDER reports the least number of compatibility issues. These differences are also significant as confirmed by a Mann-Whitney-Wilcoxon (MWW) test at a significant level[10] at 0.001.

We note that this experiment, although with a large number of apps, supports the same findings discussed previously. First, there is a strong need to invent systematic approaches to harvest compatibility issues detection rules (i.e., identifying incompatible APIs). As shown in Figure 6, the number of issues reported by CIDER and FicFinder (with manually summarized rules) is significantly less than that achieved by IctApiFinder and CiD (with systematically harvested rules). Furthermore, the fact that the intersection between the results yielded by the selected tools is quite small suggests that existing tools could be leveraged to complement each other. This result further shows that there is still a gap in the community to implement promising approaches to flag compatibility issues in Android apps, i.e., the capability of detecting compatibility issues has not been mature. Last but not the least, we believe that it is not exactly fair to directly compare existing tools targeting compatibility issues detection in Android apps as the evolution of the Android ecosystem is very fast. Tools developed at different times will likely collect a different set of incompatible APIs (e.g., the incompatible APIs collected by IctApiFinder are from 4 to 27, while CiD is from API 1 to 25), which subsequently will lead to a different set of compatibility issues. Therefore, we argue that, when comparing compatibility issues detection tools, there is a strong need to

[10]Given a significance level $\alpha = 0.001$, if p-value < α, there is one chance in a thousand that the difference between the datasets is due to a coincidence.

Just Accepted

make sure that the underlying set of incompatible APIs is kept the same, which is however non-trivial to achieve as existing tools may not always be made open-source.

RQ3 Findings

CiD is able to yield more compatibility issues than the other tools. However, their results do not overlap much, suggesting that existing tools are complementary to each other and yet still have limitations to achieve comprehensive compatibility issues detection. Furthermore, the fact that CiD and IctApiFinder can yield significantly more results than FicFinder and CIDER suggests that it is essential to leverage systematic approaches to mine incompatible APIs so as to support the comprehensive detection of compatibility issues.

5 *ANDROMEVOL* AND ITS EFFECTIVENESS EVALUATION

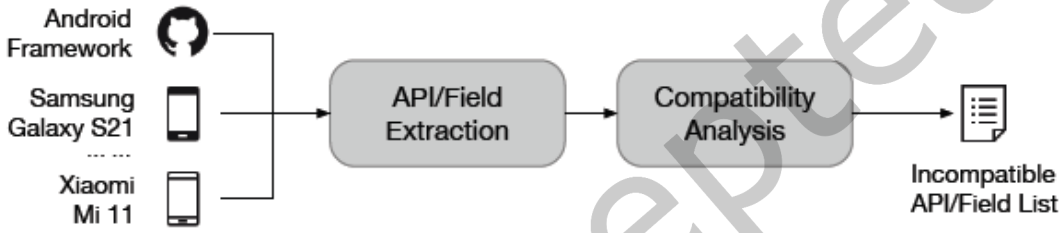

Fig. 7. The working process of *AndroMevol*.

5.1 Approach

The working process is illustrated in Figure 7. *AndroMevol* is responsible for experimentally pinpointing incompatible APIs, which could be accessed by Android apps that subsequently will suffer from compatibility issues. While existing approaches are either labor intensive or only the evolution of the official Android releases is considered, *AndroMevol* not only takes the evolution of official Android releases but also the evolution of third-party Android customizations into consideration. It takes as input a set of Android frameworks extracted from various sources (i.e., the official Android open-source code and customized ROMs released by different manufacturers) and outputs a list of incompatible Android APIs through occurrence detection. It achieves this through two key steps, (1) API Extraction and (2) Compatibility Analysis. We now detail these two steps, respectively.

5.1.1 Step 1 – API Extraction. We want to systematically identify all the incompatible Android APIs, including both device-specific APIs and evolution-induced ones. Concerning APIs, we extract methods and fields modified by the access modifier public or protected both from java classes and interfaces. Additionally, the examined APIs not only include the publicly provided ones [14] but also those restricted APIs (i.e., non-SDK APIs) [42, 76]. It is worth mentioning that we do not take the implementation of the collected APIs into account as we cannot extract the implementation of the APIs easily and we either cannot determine if the implementation update of the APIs would eventually induce compatibility issues. We will discuss the effects of such extraction on threats to validity. To this end, we need to consider as many versions of Android frameworks as possible, including the official ones mainly maintained by Google and the customized ones provided by different Android manufacturers such as Samsung and Xiaomi (i.e., hereinafter referred to as brands). For each brand, we need to consider one framework at each Android API level.

Just Accepted

In practice, this is non-trivial to achieve. First, it is expensive to purchase a complete set of physical devices covering all the brands. Yet, some brands (or specific versions of those brands) are no longer available on the market and hence cannot be purchased. Therefore, for simplicity, we will only consider the major brands and resort to their publicly released ROMs (instead of the physical phones) to locate the bytecode of Android frameworks. To determine the Android version of the released ROMs, we extract the ROMs and check the attribute of `ro.build.version.release` in the file `build.prop`. Second, during the evolution of the Android system, the framework code has been relocated to different locations or even changed into other formats that are hard to interpret.

To overcome this, we propose a best-effort approach to disclose incompatible Android APIs. The more frameworks provided, the more complete results the approach will yield. For the official Android frameworks, to ensure full coverage (one framework per API level), we resort to the official Android Open Source Project (instead of the released ROMs) to locate Android APIs. To support the analysis of both Android framework source codebase and bytecode versions (often named as *framework.jar*), we implement in this module two parsers.

Source Code Parser. This parser directly parses all the Java files located in the Android framework codebase project.[11] For each Java file, it records all its publicly defined methods and fields. Inspired by the approach proposed by Li et al. [46], we further refine the previous results by taking into account the following features:

Inheritance: A sub-class will implicitly extend all of its super classes' public or protected methods even if it does not explicitly redefine them. Similarly, when a given redefined method is dropped out in the sub-class, there is still a need to keep track of it as it could still be available (e.g., implicitly inheriting from its superclass, for which it has not yet been dropped out). These situations need to be carefully taken into account in order to achieve accurate results.

Generic type: Generic type is a Java feature that parameterizes types for convenience. An example of generic type could be the second parameter (i.e., *E*) defined in method *<LinkedList: E set(int, E)>*. This parameter (or generic type) will make it complicated to syntactically match its usages in practice, e.g., its usages could be both *set(int, String)* and *set(int, Float)*. This feature needs also to be resolved.

Varargs: Varargs is a Java feature that challenges the syntactic detection of API usage in Android apps. Varargs (defined as *TYPE...*) allows methods to receive an arbitrary number of parameters, which traditionally can be done through an array. For example, the *MessageFormat.format* method is declared as *format(String, Object...)*. The three dots after the final parameter's type indicate that the method can receive two or more parameters in practice, e.g., the actual usage of this method could be *format(String, Object)* or *format(String, Object, Object)*, etc.

Bytecode Parser. The implementation of this parser is more straightforward. Similar to the source code parser, it leverages Soot to go through all the Java classes in the jar file and record all the publicly defined methods and fields. It then goes one step deeper to take *class inheritance* into consideration to refine the results. The *generic type* and *varargs* features are ignored in this parser as they are expressed in different formats compared with the expressions in source code (i.e., the expressions such as *<LinkedList: E set(int, E)>* and *format(String, Object...)* do not exist in the decompiled Java files).

During our analysis, we noticed that the APIs extracted from the source code repository and the framework.jar bytecode have some differences, i.e., some methods in the source code repository are not included in the framework.jar, and vice versa. By comparing the source code repository with its corresponding official framework.jar (at the same API level), we realize that some packages exist only in the source code but are not packaged into the framework.jar. Similarly, many APIs extracted from the bytecode do not exist in the source code. That's because the same source code files in different releases or different customizations are packed into different package files. This difference will unavoidably impact the subsequent compatibility analysis. To mitigate such an impact, we resort to a filter to exclude packages that only exist in source code or bytecode. The filter eventually contains

[11] https://github.com/aosp-mirror/platform_frameworks_base

Just Accepted

3,148 package names, which have been subsequently applied to all the API extractions, no matter which parser is used.

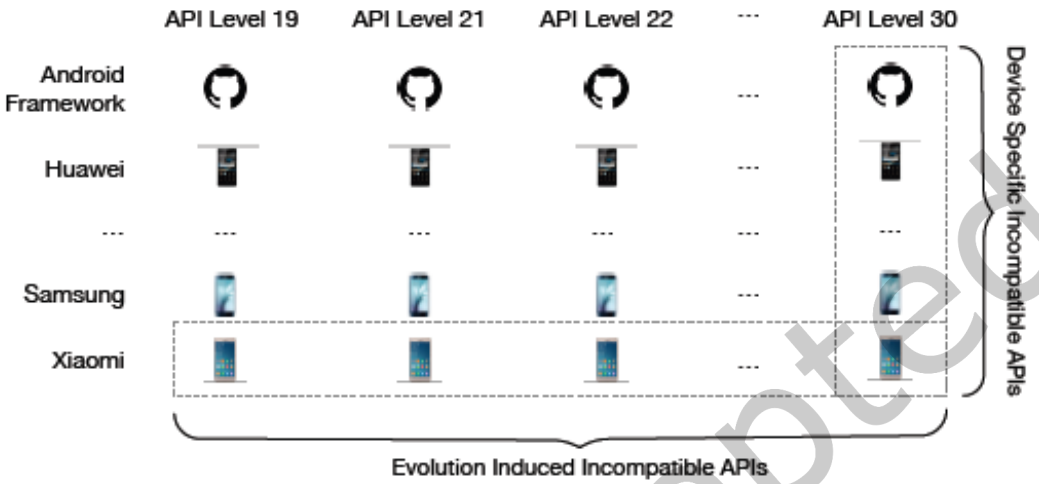

Fig. 8. Incompatible APIs identification in Compatibility Analysis.

```
1  public <U> void method1(U t1) {
2    ...

4  public <E extends java.lang.String> void
     method2(E t2) {
5    ...

7  public void method3(java.lang.String... t3) {
8    ...
```

(a) Java source code with generic types and varargs

```
1  public void method1(java.lang.Object) {
2    ...

4  public void method2(java.lang.String) {
5    ...

7  public transient void method3(java.lang.
     String[]) {
8    ...
```

(b) Jimple code generated with Soot

Fig. 9. Code example for generic types and varargs.

5.1.2 Step 2 – Compatibility Analysis. The second module of *AndroMevol* takes the extracted Android APIs as input and pinpoints potential incompatible methods and fields that could cause runtime crashes or unexpected behaviors if accessed by Android apps without appropriate guard checks. It aims at generating a complete list of incompatible APIs that could be leveraged to support existing or emerging app analysis approaches to soundly pinpoint practical compatibility issues in Android apps.

Just Accepted

Table 6. Distribution of exclusive/absent methods and fields in different API levels at different vendors. Recall that for the official platform, we have recorded the evolution-specific results for all the available Android API levels. This table only presents the results from version 19 to 30 for comparison purposes.

API level	API	Official		Huawei		Xiaomi		Oneplus		OPPO		Samsung		Total (Distinct)
		Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	Exclusive	Absent	(Methods/Fields)
19	Methods	765	21,498	1,428	16,148	324	20,478	42	17,105	1,831	15,844	12,606	5,206	22,938
	Fields	88	28,533	4,864	20,245	211	26,538	40	24,671	6,665	18,503	12,540	12,524	28,670
21	Methods	579	24,347	3,531	18,916	4,472	17,561	769	21,294	33	21,938	12,434	9,915	24,955
	Fields	25	28,140	5,758	19,560	5,569	19,280	650	24,235	18	24,815	12,693	12,372	28,165
22	Methods	585	36,669	4,010	30,604	9,071	25,083	704	33,369	5,922	28,801	13,491	21,051	37,290
	Fields	25	45,240	6,542	35,677	8,418	33,122	1,812	39,946	10,434	31,530	13,947	27,985	45,281
23	Methods	580	32,134	2,139	27,976	8,424	21,299	532	29,185	3,356	26,351	14,544	15,344	32,723
	Fields	25	39,386	3,723	32,458	6,279	29,195	2,804	32,660	6,540	28,891	15,929	19,914	39,413
24	Methods	1,518	31,851	2,777	27,478	7,949	22,100	-		233	30,074	17,445	12,957	33,371
	Fields	408	36,930	5,175	27,941	7,486	25,260			1,021	32,766	18,565	14,609	37,340
25	Methods	1,534	34,705	-		7,852	24,916	585	32,245	4,700	28,371	18,037	15,251	36,246
	Fields	415	40,324			7,538	28,399	809	35,172	8,135	27,961	18,944	17,538	40,741
26	Methods	1,155	49,485	3,978	33,442	8,343	28,835	687	36,702	4,755	48,514	18,056	19,730	55,464
	Fields	372	53,367	9,420	32,956	6,898	35,040	1,085	40,958	4,852	49,152	19,064	23,501	56,769
27	Methods	1,064	41,635	3,405	35,633	8,709	30,212	115	39,055	5,910	32,970	19,462	19,821	42,706
	Fields	347	74,642	31,195	37,716	6,594	61,938	72	68,952	9,869	58,603	20,194	48,874	74,989
28	Methods	700	61,900	4,993	52,667	15	57,390	19,471	35,732	7,343	49,928	19,843	37,828	63,629
	Fields	61	63,080	12,073	41,844	23	53,302	7,589	47,565	12,533	40,581	19,148	34,732	63,634
29	Methods	392	70,555	6,905	50,192	11,621	45,191	1,743	54,899	9,048	47,627	26,812	30,334	70,962
	Fields	36	66,569	12,052	46,243	8,556	49,232	2,051	55,630	13,965	43,715	20,934	37,321	66,605
30	Methods	443	79,750	-		13,455	43,557	2,526	54,401	11,989	57,321	28,494	28,899	111,064
	Fields	110	70,590			9,240	47,223	9,072	47,370	15,332	48,205	22,605	34,405	72,040
Total (Distinct, Device-specific)		3,473	346,673	59,805	214,247	32,548	273,885	42,456	260,937	52,939	300,003	108,082	196,486	397,678
Total (Distinct, Evolution-induced)		Exclusive: 7,374 Absent: 23,126												

The approach we take to identify incompatible APIs is represented in Figure 8. Given a method or field in a framework (either official or customized), we consider it as an incompatible API introducing compatibility issues if (1) within the same brand, it exists in some versions but not in others (the evolution induced incompatible APIs as is shown in every single row in Figure 8) or (2) at the same API level, it exists in the frameworks of some brands but not in others (the device-specific incompatible APIs as is shown in every column in Figure 8). Following this rule, this module visits every method and field in each framework to locate potential incompatible APIs, which are then put into a single configuration file for easing external usage.

Because *generic type* and *varargs* features are not involved in the bytecode parser, the results yielded by the bytecode parser will be slightly inconsistent with the ones yielded by the source code parser as we collect APIs both from source code and the file framework.jar in released ROMs. To mitigate this issue, we put additional effort into this module to unify the results before conducting the actual compatibility analysis. Figure 9 shows an example of how these features are represented in intermediate representation for compiled Java programs. Figure 9a represents the declaration of methods with generic types and varargs as parameters while Figure 9b displays the corresponding Jimple format, which is the principle Intermediate Representation and basic analysis unit in Soot [17]. Finally, the unification is done by applying the following rules to the results yielded by the source code parser.

method(T)→ method(java.lang.Object)

method(T extends java.lang.Util)→ method(java.lang.Util)

method(java.lang.String...)→ method(java.lang.String[])

The rules are built based on our observations on how *generic type* and *varargs* features are handled by the Java compiler and Soot. After applying these rules, the *generic type* and *varargs* no longer persist in the extracted Android APIs, resulting in unified results that also simplify their potential usages (i.e., no need to consider *generic type* and *varargs* features anymore).

Just Accepted

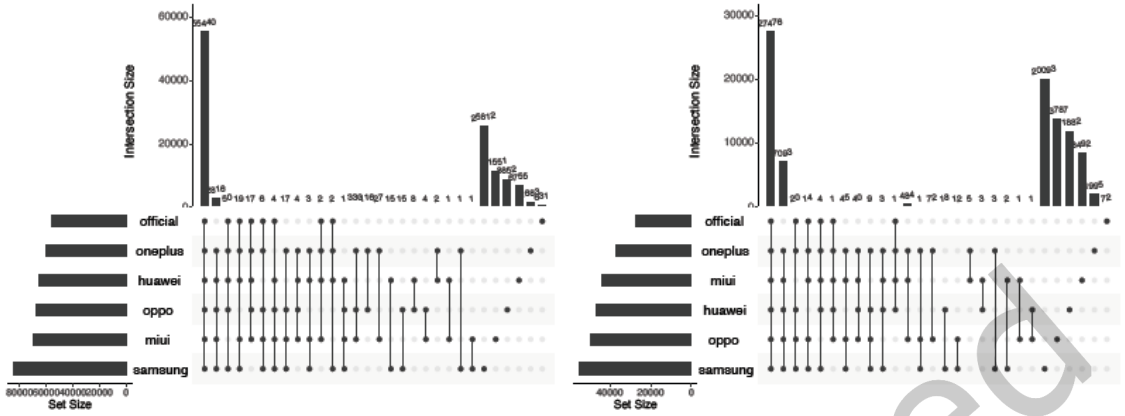

Fig. 10. UpSet plots of the intersections of methods (left) and fields (right) offered by different vendors at API level 29. An UpSet plot is made up of three parts: (1) The left barplot presents the total size of each set, (2) The bottom plot presents all the possible intersections, and (3) The top barplot presents the occurrence of each intersection marked in the bottom plot.

5.2 Effectiveness of *AndroMevol* (RQ4)

This research question concerns the effectiveness of our *AndroMevol* prototype itself in incompatible API gathering. We set out to evaluate if it is capable of automatically generating a list of incompatible Android APIs (i.e., methods and fields) for helping the community spot potential compatibility issues in Android apps.

5.2.1 Datasets.

For the sake of simplicity, in addition to the official Android system, we select five additional platform vendors for this experiment. The five selected vendors are Samsung, Xiaomi, Huawei, OPPO, and Oneplus. We chose these because they are among the most popular Android brands in the market. For the official Android system, we consider all its platform versions as they are all recorded in the open-source repository. For the remaining five vendors, we consider one platform version at each API level, ranging from 19 (i.e., the minimal version that still holds over one percent of distribution) to 30 (i.e., the latest version at the time when we conducted this study). Ideally, we should additionally consider 55 (5 platform vendors * 11 platform versions) different Android platform versions. Unfortunately, it is non-trivial to collect those frameworks related to specific vendors. Indeed, the vendors have neither maintained a complete release document nor recorded the AOSP version in their released ROMs. As a result, we have to download ROMs from different sites (notable ones include [15, 18, 19] etc.) to locate the correct ones aligned with certain AOSP versions. This step has cost the authors more than a month to complete, and in total, we have downloaded and explored over one thousand ROMs. Still, we cannot successfully extract the platform frameworks for Huawei at API levels 25 and 30 and OnePlus at API level 24. To this end, we have to ignore those versions when applying *AndroMevol* to generate the list of incompatible Android APIs. Nevertheless, ignoring the three versions should not impact the reliability of the generated incompatible API list. That's because we do not take them into consideration when we determine if APIs are incompatible or not. Moreover, these missing ROMs hardly exist as we cannot find them in the wild (even if we have spent more than one month searching). Take Huawei as an example, its latest Android OS has been replaced by its HarmanyOS. The ROM with API level 30 customized from the official Android was not even published to the public.

Just Accepted

5.2.2 Results.

Table 6 summarizes our experimental results. **In total, *AndroMevol* identifies 397,678 incompatible APIs with both evolution-induced and device-specific APIs considered that do not exist in all the considered platform versions**. This list contains 388,819 device-specific ones that only exist in some vendors' platforms but not in others and 30,500 evolution-induced ones (i.e., summarized in the last row) that only exist in some framework versions but not in others even within the same vendor. It is worth noting that the same incompatible API could not only reside in device-specific APIs but also in evolution-induced ones. In total, 21,641 incompatible APIs both belong to device-specific and evolution-induced APIs and 23,459 distinct incompatible APIs were harvested. For example, an API is added in a newer official release. Some customized OSs merge such APIs but others may not, which would introduce both device-specific and evolution-induced APIs. Every platform contains APIs that are (1) exclusive to the platform version itself and (2) absent from the other platform versions at the same API level. This experimental result suggests that both the evolution of the Android framework and the customization of the official framework introduced by different Android vendors have introduced divergences among the different platform versions. Since a given Android app is expected to be installed and executed on all those platforms, such divergences may cause inconsistencies in offered Android APIs and, subsequently, cause app crashes on some platforms while being successful on others (i.e., compatibility issues). Additionally, it is worth noting that the number of incompatible APIs is increasing as the evolution of Android releases including both the official and customized ones, which would induce more compatibility issues if the developers do not handle such incompatible APIs properly during their development. To avoid such potential issues in released apps, developers should pay more special attention to their development while invoking such APIs. Moreover, OS maintainers could proactively reduce the number of incompatible APIs in their releases.

To evaluate the correctness of the harvested incompatible APIs, we resort to human efforts to check a set of sampled results manually. We first randomly selected 378 incompatible APIs from the selected detection tools, CiD, FicFinder, CIDER, and IctApiFinder as well as the incompatible API harvesting tool, Pivot [12], and then manually checked their incompatibility to set them as ground truth. With the validated ground truth, *AndroMevol* achieved a high precision of 82.24% and recall of 96.17% in identifying incompatible APIs. What's more, the incompatible APIs missed by *AndroMevol* are almost harvested from other packages, such as the external Unicode support [13], which is out of consideration of *AndroMevol* and we will try to resolve in future.

Figure 10 further illustrates with an example the incompatible methods/fields differences (via respectively two UpSet plots) between different platforms at API level 29. The left barplot in the UpSet plot represents the total number of elements from different sets. For example, the left UpSet plot in Figure 10 shows that Samsung provides the highest number of methods for developers while official gives out the lowest number of methods, which are represented in the left barplot with the number of methods over 80,000 and 60,000, respectively. The top barplot shows the detailed number of elements common or unique among different sets, such as the first bar with the number 55,440, which corresponds to the direct bottom dotted line and represents that the total number of 55,440 methods are common among all the six sets, and the last bar with number 631 representing that the official framework provides 631 unique methods comparing with other five vendors since there exists only one dot in the bottom plot. Clearly, and as expected, the majority of methods and fields are kept for all the considered platforms, while each of them has some exclusive ones. The fact that different combinations of intersections (for both methods and fields) exist **shows that there are indeed divergences among the selected platforms, which can lead to potential compatibility issues**. We observe that, perhaps surprisingly, **there are a significant number of methods and fields that are not available in the official framework**

[12] In total, we have 23,459 distinct incompatible APIs harvested and resort to the famous Sample Size Calculator [16] with a confidence level of 95% and a margin of error of 5%. It gave out a sample size of 378 based on the total sample size of 23,459. We, therefore, selected 378 incompatible APIs.

[13] https://developer.android.com/guide/topics/resources/internationalization

Just Accepted

but are available on all the other platforms. Our in-depth investigation reveals that this phenomenon is caused by the fact that some APIs have been removed by the official framework during its evolution but are kept by customized frameworks. This suggests that **the customized framework developers do not always keep up with the updates of the official framework**. This confirms our previous finding [49] that there are divergences between the official Android framework and the selected customized frameworks.

RQ4 Finding

AndroMevol is effective in harvesting incompatible APIs. Among the official Android framework and its five branches, it has identified a total of 397,678 incompatible APIs, giving an accuracy of 98%.

5.3 Comparison with state-of-the-art tools in incompatible API gathering (RQ5)

The compatibility issue detection tools we selected for our replicability study all have their approach to harvesting incompatible APIs. It should be noted that these detection tools were developed first to gather incompatible APIs and then to feed such APIs to their analysis implementation to pinpoint compatibility issues. In addition, the identified compatibility issues are only induced by a subset of the collected incompatible APIs as we cannot enumerate all of these incompatible APIs in our collected finite app dataset. In contrast, our tool *AndroMevol* was designed to systematically identify incompatible APIs with both evolution-induced and device-specific ones considered. In our last research question to evaluate the performance of our approach, we propose to further compare the ability to identify incompatible APIs between our tool and the state-of-the-art detection tools – FicFinder, CIDER, CiD, and IctApiFinder. Our comparison study aims to address the following concerns: How many incompatible APIs are identified by these detection tools? Could the APIs pinpointed by our tool cover the incompatible APIs identified by other detection tools? If not, what is the reason for the differences in identifying the incompatible APIs?

Besides the detection tools we discussed in the previous comparative study, we re-involved the excluded API gathering-oriented tool, **Pivot** [72]. Pivot, different from the previous detection tools aiming to detect compatibility issues in apps with different approaches, is proposed to automatically learn Fragmentation-Induced API-correlations from released Android Apps. The tool first builds inter-procedural control flow graphs by associating the app's call graph and every method's control flow graph and then traverses the whole graph to identify device-checking statements and evaluates the conditions on each branch. With the identified device-checking statements and the constraints on each branch, all reachable APIs with the device constraints are constructed as API-device correlations. At last, the paper resorts to two metrics: in-app confidence and occurrence diversity, to filter out invalid API-device correlations. The API-correlation is a tuple comprising of an API invocation and the guarded device-checking conditions, such as `<android.hardware.Camera$Parameters: void setRecordingHint(boolean)>, “Nexus 4”>`.

5.3.1 Datasets.

We extracted the incompatible APIs used in the selected four different detection tools, including CiD, IctApiFinder, CIDER, and FicFinder, and downloaded the incompatible APIs generated by the API harvesting tool Pivot. We involve the tool, Pivot, back because it was developed for incompatible APIs gathering from the published Android apps rather than issue detection, which we discussed in RQ3 and excluded it. We now detail the differences between the selected issue detection tools including CiD, IctApiFinder, CIDER, and FicFinder as well as Pivot and *AndroMevol* with regard to the ability in collecting incompatible APIs.

5.3.2 Results.

Figure 11 represents the differences between the selected detection tools, including the newly considered API harvesting tool Pivot, and our proposed tool *AndroMevol*. In the UpSet plot, the left bar chart shows how many incompatible APIs were pinpointed by the detection tools. This bar chart clearly shows that our tool *AndroMevol*

Just Accepted

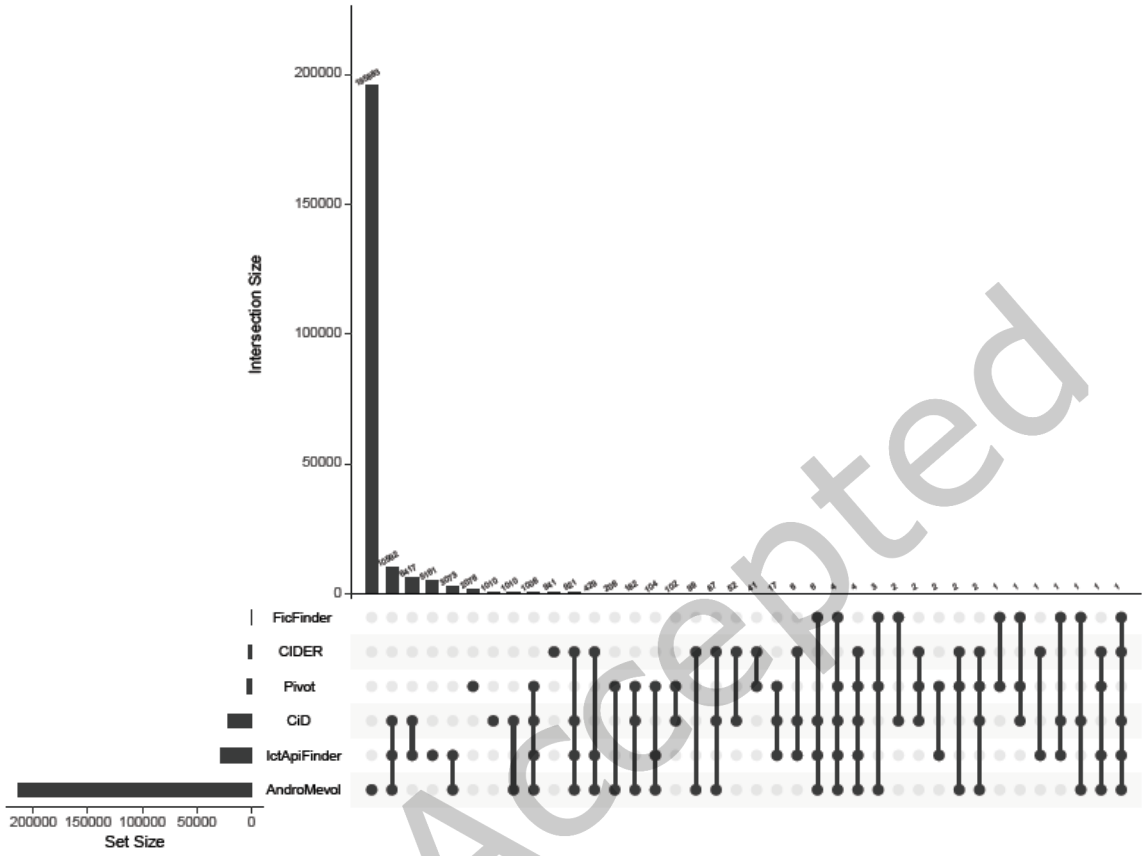

Fig. 11. UpSet plot of incompatibility APIs among different detection tools. An UpSet plot is made up of three parts: (1) The left barplot presents the total size of each set, (2) The bottom plot presents all the possible intersections, and (3) The top barplot presents the occurrence of each intersection marked in the bottom plot.

Just Accepted

(with reporting 236,099 incompatible APIs) outperforms the other detection tools by pinpointing many more incompatible APIs.

The possible intersections represented in the top and bottom plots further reveal that most of the incompatible APIs identified by *AndroMevol* were actually missed by the other selected state-of-the-art detection tools. As illustrated in the first column of the vertical bar chart, 195,883 incompatible APIs are exclusively harvested by *AndroMevol*, accounting for 82.97% of the total number of harvested incompatible APIs. This result is expected as *AndroMevol* not only systematically records the historical changes of the official Android framework but also takes into account the various customizations done by five smartphone manufacturers. It is still worth mentioning that even though the majority of the collected incompatible APIs are unique to *AndroMevol*, *AndroMevol* still could cover a considerable number of incompatible APIs reported by our selected tools. For example, the number of incompatible APIs in the intersection among AndroMevol and CiD accounts for 82.70% (13,938/16,694) with regard to the incompatible APIs given by CiD.

Interestingly, even though *AndroMevol* significantly outperforms the other tools by harvesting much more incompatible APIs, it does miss some incompatible APIs within all sorts of types that are identified by the selected state-of-the-art tools. We, therefore, go one step further to manually check those incompatible APIs harvested and utilized in the published tools and investigate why they cannot be identified by our tool *AndroMevol*. Our analysis shows the main reason is that the different approaches do have different Android eco-system coverage. *AndroMevol* mainly focuses on incompatible APIs with different signatures from the Android framework, but other tools except FicFinder take different Android packages and method semantics into consideration. We now discuss the identified differences in detail.

- FicFinder was the earliest detection tool to identify compatibility issues. It harvested incompatible APIs manually and contains only 20 incompatible APIs. The bottom plot also reveals that our tool *AndroMevol* could also cover most of the manually collected APIs but not all of them. To identify incompatible APIs, *AndroMevol* extracts incompatible APIs from *framework.jar* provided in Android ROMs, which mostly contains package of Android framework but not others. Since we do not have a consistent correspondence between Jar files in Android ROMs and their source code, it is non-trivial to extract APIs for other packages from other Jar files in Android ROMs. Therefore, we only took framework into consideration when we developed *AndroMevol*, which might miss some incompatible APIs provided in other Jar files.
- Pivot [72] was proposed to collect the API correlations by learning from real published Android apps to empower their compatibility issue detection tool (i.e., FicFinder [71]). Our in-depth analysis reveals that Pivot considers more than 1,000 different device-checking conditions, such as Vivo, Lenovo, Meizu, and HTC-related device checks, etc., which were not taken into account in our experiments (i.e., we only considered five brands at the moment). This result sheds light on our future work to consider more popular Android devices for harvesting an even more comprehensive set of incompatible APIs.
- CIDER focuses on compatibility issues triggered by invoking callback APIs. Some of the callback APIs are added during the evolution of Android, which can also be identified by *AndroMevol*. Some other incompatible APIs were detected because they have involved semantic changes (the signature has not been altered), which, by far, have been overlooked by our tool *AndroMevol* (cf., 6.2.1).
- CiD, the state-of-the-art compatibility issue detection tool, extracted incompatible APIs from the provided API summary in the AOSP source set. The summary contains APIs from not only the Android framework but also other packages. Unfortunately, the API summary is no longer provided for the newer Android releases. We, therefore, harvested incompatible APIs from the Android framework only (extracting APIs both from the source code of Android framework and *framework.jar* provided in Android ROMs) in our approach, resulting in some incompatible APIs missed by *AndroMevol* (i.e., only being reported by CiD). For example, APIs from class `android.net.wifi.p2p.WifiP2pDevice` are provided in the summaries but we cannot extract them from the source code of the Android framework. That's because the java file locates under the source set `packages` [14] rather `framework`. We focus on the Android framework only as it provides the basic services that Android apps directly and heavily rely on [22, 28, 41, 46, 48, 54, 75]. We do not extend the API levels support of CiD here because we focus on the comparison of incompatible APIs gathering between the collected tools and our proposed artifact. In addition, we also adopt the same approach focusing on the Android framework to harvest incompatible APIs (as CiD^n in Section 6.1) since the API summary is no longer provided in the AOSP source set.
- IctApiFinder, even though the module of compatibility issue detection has been updated, the module of incompatible API extraction is still the same as the one published in the corresponding paper. To retrieve incompatible APIs, the developers extracted APIs from released SDKs API level 4 to API level 27. The SDK

[14]https://cs.android.com/android/platform/superproject/+/master:packages/modules/Wifi/framework/java/android/net/wifi/p2p/WifiP2pDevice.java

Just Accepted

contains all of the APIs necessary for App development provided by Google. Our tool extracts only from the Android framework. Thus, the APIs not in the framework are missed by *AndroMevol* (cf., 6.2.1).

RQ5 Findings

Compared with the state-of-the-art detection tools in collecting incompatible APIs, *AndroMevol* outperforms all of them by harvesting at least eight times more incompatible APIs, including both device-specific and evolution-induced ones and 195,883 previously unreported ones.

6 DISCUSSION

We now discuss the key implications of this research, including prioritized research directions that should be conducted for mitigating the fragmentation impact on the Android community. Our literature review and experimental findings raise a number of issues and opportunities for research and practice communities.

6.1 Implication

Comprehensive Compatibility Issues Detection: Our systematic literature review revealed that app compatibility issues are induced by five types of incompatible APIs. Existing detection approaches all focus on compatibility issues induced by some specific types of incompatible APIs. To pinpoint issues induced by all types of incompatible APIs, customers need to resort to different detection tools, which makes the detection laborious and arduous. Therefore, we argue for a comprehensive issue detection approach, which could detect issues induced by all types of incompatible APIs.

Continuous Improvement to Adapt to the Fast Evolution of the Mobile Ecosystem. With the rapid evolution of the open-source Android Operating System, detection tool maintainers need to take new system releases into account. Many device vendors release lots of different models as their own publishing step. To detect the newly introduced compatibility issues, these tools need to be refined once a new system version is released and a new device induced. However, these tools are not self-adaptive. They all need to be carefully adjusted.

As an example towards demonstrating the necessity to continuously update the tools to adapt to the fast evolution of the mobile ecosystem, we spend additional efforts to update the open-source CiD project by extending its supported API ranges from 1-25 to 1-31 (Android12 with API level 31 is the latest Android release). The updated version is referred to as CiD^n. We then apply CiD^n to analyze the apps in Dataset2. Figure 12 summarizes the experimental results, along with that achieved by the original CiD. Clearly, CiD's performance has indeed been improved after adapting to the latest release of Android frameworks. This evidence strongly suggests the necessity to keep adapting compatibility issues detection tools to support the latest changes of the mobile ecosystem. We therefore argue that different automation approaches are needed to facilitate the extraction of Android APIs in order to automate issue detection when new Android versions and devices are released.

Integrating Dynamic Testing to Verify Compatibility Issues: Currently, most research approaches proposed to tackle compatibility issues in Android apps rely on static analysis. However, efficient, static analysis is also known to yield many false-positive results. We argue that dynamic testing approaches should also be included to supplement the analysis of static analysis approaches (e.g., to practically verify the results yielded by static analysis approaches). It is nevertheless non-trivial to build a comprehensive dynamic testing environment for checking compatibility issues, as it needs to include all publicly available Android devices, for which the number is also continuously changing. To cope with this, we argue that crowdsourced mobile app testing could be leveraged, especially in lightweight mode directly supported by the Android system, to pinpoint and subsequently mitigate compatibility issues.

Characterizing Semantics-changing Incompatible APIs: In addition to the five types of incompatible APIs discussed in this work, which are all related to the existence of the APIs, there is another type of API-induced

Just Accepted

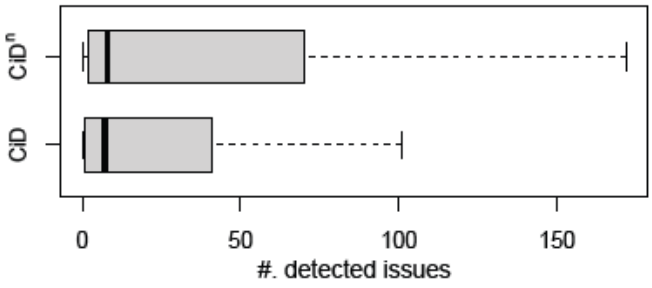

Fig. 12. Comparison between original CiD and API life-cycle extended CiD.

compatibility issue that goes beyond APIs' existence to concern their semantic changes. Given an API with semantic changes, even if its signature persisted in the framework, the client apps accessed into it could also be impacted. Such semantics changes will be propagated to the client app, which may not have yet adapted to such changes. As recently revealed by Liu et al. [50], there are indeed a number of Android APIs involving semantic changes during the evolution of the framework. However, such semantic changes are hard to be automatically identified, so as to the corresponding compatibility issues. Therefore, we argue that our community should also pay special attention to semantics-changed incompatible APIs and invent advanced approaches to mitigate them, either by carefully (1) documenting them if it is unavoidable to change the semantics of existing APIs or (2) testing the client apps to identify and fix such issues before publishing the apps to end-users.

Supporting Automated Compatibility Issue Repair: Finally, after API compatibility issues are identified, we argue that automated approaches are also needed to help developers fix them [79]. This is especially true for such apps that have already been released to the public, as users may not even be able to install the apps or face runtime crashes even if the apps can be successfully installed. Automated repairing approaches could keep users from encountering such unfavorable situations, meanwhile helping app developers fix the issues for better future releases.

Supporting Issue Detection Besides Android Ecosystem: Compatibility issues exist in all sorts of different systems besides Android, such as the highly evolved Linux and their third-party customizations as well as the web browser systems. Different APIs are provided along with their system release. To enhance the robustness of such systems, developers should detect such issues as much as possible before their system's release. Our proposed approach could be extended to gather incompatible APIs, including the evolution-induced APIs and device-specific ones in order to do issue detection as the general issue detection approach always starts with incompatible APIs harvest. Our approach could facilitate researchers to focus on detailed algorithms to detect compatibility issues with the help of the collected incompatible APIs.

6.2 Threats to Validity

6.2.1 External Validity. There are several threats to validity associated with the results we presented in our replication study. One threat is the configuration of all our selected detection tools. All selected tools are implemented on top of the Soot static analysis framework, which also requires Android frameworks as input parameters. However, the version of the Soot and Android frameworks accessed in the selected artifacts may be still different because we cannot know the exact versions leveraged in every detection tool. Different Soot versions were released with different bug fixes and enhancement modules merged. They could have different performances and give different analysis results, breaking the analysis validity. To mitigate this threat, we meticulously align the

Just Accepted

configurations among them as much as we can to provide approximately the same environment. Another threat depends on the approach to harvest incompatible APIs, especially between IctApiFinder and CiD. IctApiFinder extracts APIs from Android framework API levels 4 to 27 based on published artifacts, while CiD acquires from the source code of Android framework API levels 1 to 25 on their approach. Different ranges of API levels and the trade-offs made while pinpointing incompatible APIs would unavoidably bring in discrepancies, which may result in different performances even on the same dataset. In addition, our approach focuses on the signature of the APIs while CIDER could identify APIs with semantic changes, which are missed by *AndroMevol*.

The major threat to the validity of the work to extract a complete set of incompatible APIs is related to the selection of vendors with customized Android frameworks, which may not be representative of the whole ecosystem as there are many more vendors available in the ecosystem. Nevertheless, we have attempted to mitigate this impact by focusing on the most popular Android brands. Furthermore, because of various challenges put on by both Android and the customized vendors, we cannot locate the correct ROMs on the internet, although we have spent a significant amount of time doing that. Even for identified ROMs, we may not be able to extract the framework code for some vendors' framework platform versions, making the results incomplete. The corresponding results do not reflect the whole set of incompatible APIs in the wild. In addition, we focus the API extraction on the package framework.jar, which does not contain all the available APIs (such as some incompatible ones only identified by CIDER, IctApiFinder, and CiD, etc.). That's because we cannot locate the other package names and locations providing other APIs among different vendor OS releases. However, we have made extra manual efforts to ensure that the results that can be computed are indeed correct.

6.2.2 Internal Validity. Since we want to include a complete evolutionary history of the official Android framework, which is essential to observe evolution-induced incompatible APIs, we resort to two different approaches to extract APIs (e.g., source code parsing and direct bytecode extraction). These two approaches may introduce inconsistencies as (1) we cannot locate the exact framework version (at the source code level) that is customized by the third-party vendors, and (2) not all the source code files (e.g., systemui-related code is not included in the framework bytecode but compiled into an independent APK) are packaged to the final framework version embedded in real devices. To mitigate the potential impact, we collect APIs from the source code of the official Android releases and harvest APIs from the same API level decompiled framework.jar extracted from the official Android virtual machines provided in Android Studio. We collect the package names that exist in both source code and framework.jar and set the list of package names as an API filter (cf., 5.1.1). With the API filter, we exclude APIs that do not reside in our collected packages (i.e., the package names of the APIs do not exist in our collected package names). This process, unfortunately, could also exclude actual incompatible APIs. However, the number of excluded APIs is generally small, making this impact neglectable.

We only take the API signature (the method signature refers to the combinations of the method return type, the method name, and the method parameters type list, thus method's checked exceptions are ignored.) into consideration when we determine incompatible APIs (cf., 5.1) in our proposed *AndroMevol*, which would induce false negatives. Besides, the updates of the class hierarchy including member methods and fields movement among parent and child classes are also ignored in our implementation. However, the movements of the APIs (methods/fields) among parents and their child classes can be and should be handled by the detection tools, such as the implementation of the detection tool, CiD, so as to avoid an explosion of the number of incompatible APIs. If we add all APIs belonging to parents' class to children's class, the number of incompatible APIs would increase sharply. To have a fair number of incompatible APIs and take full advantage of detection tools, we hand over the capability to manage class hierarchy to detection tools. In the future, we would enhance or develop our artifacts to consider these limitations to systematically detect as many types of compatibility issues as possible.

Just Accepted

7 RELATED WORK

In recent years, compatibility issues have been a hot topic in the Android community [35, 39, 59, 61, 68, 79]. Since the apps are inseparable from the official Android APIs, it is essential to probe compatibility issues caused by the evolution of the Android operating systems.

Besides the tools we investigated in the paper, there are many other works handling various API issues. For example, Li et al. [46, 47] build a prototype tool, CDA, to characterize deprecated Android APIs by mining the evolution of the Android framework. Similar method has also been applied to characterize inaccessible APIs [42] and inconsistent release time of Android apps [41]. Scalabrino et al. [61] introduce ACRYL, learning from the change histories of other apps in response to API evolution. It can identify compatibility issues, yet in addition suggest repairs. The authors empirically compare ACRYL and CiD and track down no obvious winner, but the results indicate the possibility of combining the two methods in the future. Later on, they extend their work [62] by enlarging the datasets and adding some interviews and details, but there is no obvious improvement in terms of the detection approach. Xia et al. [74] conduct a large-scale study on the practice of handling OS-induced API compatibility issues and their solutions, and they propose a tool named RAPID to ascertain whether a compatibility issue has been resolved. Mobilio et al. [55] acquaint a tool named FILO which can assist Android designers in tackling backward compatibility issues caused by API upgrades. FILO is designed to recognize app methods that need to be altered to adapt to the API changes and report symptoms observed in failed executions to facilitate repair. Mahmud et al. [53] propose ACID, an approach to detecting compatibility issues caused by API evolution. Experimental results demonstrate that ACID is more accurate and faster in detecting compatibility issues than previous techniques. The fly in the ointment is that ACID only considers the changes in Android method invocations and callbacks brought about by evolution rather than considering device-specific compatibility issues.

To detect such compatibility issues, different information flows are needed to identify by constructing inter- and intra-procedural control flow graph [44]. Qiu et al. [57] did an extensive comparison among three most prominent static analysis tools including FlowDroid [21] combined with IccTA [40], DroidRA [43, 66], AmanDroid [69, 70], and DroidSafe [30]. They spotted out the advantages and shortcomings of each tools and revealed that it is important to provide detailed configuration and setup environment specification to guarantee the replicability of experiments.

In non-Android communities, research on compatibility issues is also pervasive [25, 33, 58, 60, 80]. Sawant et al. [60] analyze clients of popular third-party Java APIs and the JDK API and publicise a large dataset; also, they look into the connection between the client's response patterns and the deprecation policy the related API adopted. Chen et al. [27] present an approach named DeBBI, which leverages the test suites of various client projects to detect library behavioral backward incompatibilities.

To compare different tools developed for the same issue, Su et. al. [65] did an extensive comparison and proposed a new benchmark called Themis facilitating our research community for automated GUI testing. They collected critical bugs reported on Github with respect to their bug label revealing the severity and did experiments with five state-of-the-art testing tools integrated with Monkey [12], and then gave out qualitative and quantitative analysis result. They successfully identified 5 different challenges that these tools still face, such as the reachability of deep use scenarios, test input generation etc., and shed lights on future research based on their systematic analysis results, such as integrating heuristics to improve the capability to spot GUI bugs.

8 CONCLUSION

In this paper, we have conducted a literature review on research targeting Android app compatibility issues. Based on this review, we are able to identify nine state-of-the-art works proposed to detect compatibility issues in Android apps and among which we have summarized five types of incompatible issues reported by our fellow

Just Accepted

researchers. We then confirm the reproducibility of the selected tools based on a replication study by running the tools against their original datasets. We further go one step deeper to conduct an empirical comparison study among the selected tools. Our findings indicate that compatibility issues detection is still at an early stage, which requires attention from the community to keep improving so as to achieve sound compatibility issues detection. As categorised and reported by other researchers, there are five types of incompatible APIs available in the mobile ecosystem but none of the existing harvest approaches is capable of collecting all of them. To fill this gap, in this work, we propose to introduce to the community a novel prototype called *AndroMevol*, which endeavors to construct a list including as many incompatible APIs as possible. Experimental results demonstrate that *AndroMevol* is effective in achieving its purpose to generate a list of incompatible APIs, which are useful for supporting the detection of compatibility issues in real-world Android apps.

9 ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments on the conference version of this extension. This work was supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP200100020.

REFERENCES

[1] 2021. ACID. https://github.com/TSUMahmud/acid.
[2] 2021. ACRyL. https://github.com/intersimone999/acryl.
[3] 2021. CiD. https://github.com/lilicoding/CiD.
[4] 2021. CIDER. https://github.com/cideranalyzer/cideranalyzer.github.io.
[5] 2021. Download Pivot. https://ficissuepivot.github.io/Pivot/.
[6] 2021. FicFinder Project Homepage. http://sccpu2.cse.ust.hk/ficfinder/.
[7] 2021. IctApiFinder. https://github.com/DongjieHe/IctApiFinder.
[8] 2021. *OPPO's share of smartphone shipments worldwide.* https://www.statista.com/statistics/628545/global-market-share-held-by-oppo-smartphones/
[9] 2022. AnkiDroid. https://github.com/ankidroid/Anki-Android.
[10] 2022. Gadgetbridge. https://github.com/Freeyourgadget/Gadgetbridge.
[11] 2022. LibreTorrent. https://github.com/proninyaroslav/libretorrent.
[12] 2022. Monkey. http://developer.android.com/tools/help/monkey.html.
[13] 2022. MozStumbler. https://github.com/mozilla/MozStumbler.
[14] 2022. *Official Android API reference.* https://developer.android.com/reference
[15] 2022. Oppo firmware update site. https://support.oppo.com/au/software-update/.
[16] 2022. *Sample Size Calculator.* https://www.surveysystem.com/sscalc.htm
[17] 2022. *Soot Framework.* http://soot-oss.github.io/soot/
[18] 2022. Various firmware hosting site. https://firmwarefile.com/.
[19] 2022. Xiaomi firmware update site. https://c.mi.com/global/miuidownload/index.
[20] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Vol. 8. Addison-Wesley Reading.
[21] 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.
[22] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. The impact of api change-and fault-proneness on the user ratings of android apps. *TSE* 41, 4 (2014), 384–407.
[23] 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.* 3–8.
[24] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software* 80, 4 (2007), 571–583.
[25] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. 2016. Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 360–369.

Just Accepted

[26] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A large-scale study of application incompatibilities in android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 216–227.
[27] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124.
[28] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China). Association for Computing Machinery, New York, NY, USA, 204–215.
[29] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[30] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.
[31] Todd J Green, Molham Aref, and Grigoris Karvounarakis. 2012. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry: Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*. Springer, 1–8.
[32] 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 167–177.
[33] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? The Pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 251–260.
[34] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.
[35] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. 2019. Semantic Patches for Java Program Transformation (Experience Report). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[36] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Citeseer.
[37] Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2019. Mimic: UI compatibility testing system for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 246–256.
[38] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).
[39] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 601–614.
[40] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.
[41] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2018. MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*.
[42] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*.
[43] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*.
[44] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).
[45] 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*. 153–163.
[46] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 254–264.
[47] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. Cda: Characterising deprecated android apis. *Empirical Software Engineering* (2020), 1–41.
[48] 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 9th joint meeting on foundations of software engineering*. ACM, Saint Petersburg, Russia, 477–487.
[49] Pei Liu, Mattia Fazzini, John Grundy, and Li Li. 2022. Do Customized Android Frameworks Keep Pace with Android?. In *The 19th International Conference on Mining Software Repositories (MSR 2022)*.
[50] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. 2021. Identifying and Characterizing Silently-Evolved Methods in the Android API. In *The 43rd ACM/IEEE International Conference on Software Engineering, SEIP Track (ICSE-SEIP 2021)*.

Just Accepted

[51] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. 2022. Automatically Detecting API-induced Compatibility Issues in Android Apps: A Comparative Analysis (Replicability Studies). In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*.

[52] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2021. Deep learning for android malware defenses: a systematic literature review. *arXiv preprint arXiv:2103.05292* (2021).

[53] Tarek Mahmud, Meiru Che, and Guowei Yang. 2021. Android Compatibility Issue Detection Using API Differences. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 480–490.

[54] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*. IEEE, Eindhoven, Netherlands, 70–79.

[55] Marco Mobilio, Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. 2020. FILO: FIx-LOcus localization for backward incompatibilities caused by Android framework upgrades. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1292–1296.

[56] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. 2021. AndroidCompass: A Dataset of Android Compatibility Checks in Code Repositories. *arXiv preprint arXiv:2103.09620* (2021).

[57] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–186.

[58] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[59] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

[60] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2018. On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. *Empirical Software Engineering* 23, 4 (2018), 2158–2197.

[61] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 288–298.

[62] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Valentina Piantadosi, Michele Lanza, and Rocco Oliveto. 2020. API compatibility issues in Android: Causes and effectiveness of data-driven detection techniques. *Empirical Software Engineering* 25, 6 (2020), 5006–5046.

[63] Md. Shamsujjoha, John Grundy, Li Li, Hourieh Khalajzadeh, and Qinghua Lu. 2021. Developing Mobile Applications via Model Driven Development: A Systematic Literature Review. *Information and Software Technology (IST)* (2021).

[64] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for fast and easy program analysis. In *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Springer, 245–251.

[65] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 119–130.

[66] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2020. Taming Reflection: An Essential Step Towards Whole-Program Analysis of Android Apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2020).

[67] 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*. 214–224.

[68] Haoyu Wang, Hongxuan Liu, Xusheng Xiao, Guozhu Meng, and Yao Guo. 2019. Characterizing Android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 280–292.

[69] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1329–1341.

[70] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

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

[72] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 878–888.

[73] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1176–1199.

[74] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *2020 IEEE/ACM*

Just Accepted

42nd International Conference on Software Engineering (ICSE). IEEE, 886–898.

[75] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. 2018. How Do Android Operating System Updates Impact Apps?. In *MobileSoft* (Gothenburg, Sweden). ACM, New York, NY, USA, 156–160.

[76] Shishuai Yang, Rui Li, Jiongyi Chen, Wenrui Diao, and Shanqing Guo. 2022. Demystifying Android Non-SDK APIs: Measurement and Understanding. (2022).

[77] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 89–99.

[78] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering* (2021).

[79] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards Automatically Repairing Compatibility Issues in Published Android Apps. In *The 44th International Conference on Software Engineering (ICSE 2022)*.

[80] Jing Zhou and Robert J Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 266–277.

Just Accepted