

# Automatically Detecting API-induced Compatibility Issues in Android Apps: A Comparative Analysis (Replicability Study)

Pei Liu  
Pei.Liu@monash.edu  
Monash University  
Australia

Yanjie Zhao\*  
Yanjie.Zhao@monash.edu  
Monash University  
Australia

Haipeng Cai  
Haipeng.Cai@wsu.edu  
Washington State University, Pullman  
United States

Mattia Fazzini  
mfazzini@umn.edu  
University of Minnesota  
United States

John Grundy  
John.Grundy@monash.edu  
Monash University  
Australia

Li Li†  
Li.Li@monash.edu  
Monash University  
Australia

## ABSTRACT

Fragmentation is a serious problem in the Android ecosystem. This problem is mainly caused by the fast evolution of the system itself and the various customizations independently maintained by different smartphone manufacturers. Many efforts have attempted to mitigate its impact via approaches to automatically pinpoint compatibility issues in Android apps. Unfortunately, at this stage, it is still unknown if this objective has been fulfilled, and the existing approaches can indeed be replicated and reliably leveraged to pinpoint compatibility issues in the wild. We, therefore, propose to fill this gap by first conducting a literature review within this topic to identify all the available approaches. Among the nine identified approaches, we then try our best to reproduce them based on their original datasets. After that, we go one step further to empirically compare those approaches against common datasets with real-world apps containing compatibility issues. Experimental results show that existing tools can indeed be reproduced, but their capabilities are quite distinct, as confirmed by the fact that there is only a small overlap of the results reported by the selected tools. This evidence suggests that more efforts should be spent by our community to achieve sound compatibility issues detection.

## CCS CONCEPTS

• **Software and its engineering** → **Empirical software validation**; • **General and reference** → **Empirical studies**.

## KEYWORDS

Android, Android API, Compatibility Issue, Replication

\*The first and second authors contributed equally to this research.

†Corresponding author.

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 ACM 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](mailto:permissions@acm.org).

ISSTA '22, July 18–22, 2022, Virtual, South Korea

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9379-9/22/07...\$15.00

<https://doi.org/10.1145/3533767.3534407>

## ACM Reference Format:

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 Study). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3533767.3534407>

## 1 INTRODUCTION

Fragmentation has been a severe problem for the Android ecosystem for years. This refers to the fact that there are 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 certain Android versions with certain device features (i.e., the apps crash on other devices), leading to so-called compatibility issues.

Compatibility issues have been considered one of the most severe problems in the Android ecosystem. 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 even if successfully installed. This results in poor user experience not only for the app per se but also the whole Android ecosystem. 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, who must take into account a dizzying number of devices and OS versions, which are non-trivial and yet expensive to achieve without a proper infrastructure in place.

To address these issues, there has been a great deal of research in analyzing the compatibility issues of Android apps. In the area of static analysis, researchers have proposed various automated approaches to pinpoint one of the most common compatibility issues: API-induced compatibility issues. For example, Li et al. [35] 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. Wei et al. [57] 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.

However, it is still unclear what the status quo of Android app compatibility analyses is, what their strengths and weaknesses are, and to what extent they are able to identify all the possible incompatible Android APIs and their induced compatibility issues in real-world Android apps. Furthermore, it is also unknown to what extent can we reproduce their experimental results and how well do the tools compare with each other in terms of detecting compatibility issues. Specifically, in this work, we formulate these concerns into three research questions that we aim to answer through empirical evidence and experimental results. The three research questions are summarized as follows.

- **RQ1: What is the status quo of Android compatibility issues detection approaches?**

We propose to answer this research 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 to 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 Override/Callback. Unfortunately none of the existing approaches can tackle all five types of API-induced compatibility issues. The most recent, ACID [40], can only handle three out of the aforementioned five types.

- **RQ2: Can we replicate the experimental results yielded by state-of-the-art tools targeting compatibility issue detection?**

Replicability study has been regarded as an essential method to confirm the reliability of existing research (including both experiments and datasets) and hence has been considered an important field in the software engineering community. In the second research question, we aim to confirm the reliability of existing compatibility issues detection approaches by reproducing their experimental results against their original datasets.

Our experimental results show that the majority of experimental results could indeed be reproduced. The remaining small number of inconsistent results (yielded by IctApiFinder and FicFinder) are mainly caused by unnecessary updates of the tools (such as dependency fixes) and apps (due to unrecorded Github version of the apps).

- **RQ3: How well do the tools compare with each other?**

To answer this question and to make a fair comparison, we launch the selected 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 [42]. 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.

**Open source.** The source code and datasets are all made publicly available in our artifact package via the following link:

<https://zenodo.org/record/6516441>

## 2 STATUS QUO UNDERSTANDING (RQ1)

Towards checking how far we are in automating compatibility issues detection in Android apps, we performed a systematic literature review to understand the status quo about Android app compatibility analyses.

### 2.1 Literature Review

Figure 1 illustrates the working processes of the literature review summarized based on the guidelines provided by Keele [26] and Brereton et al. [15], as well as lessons learned from our recent practices [28, 39, 49, 61].

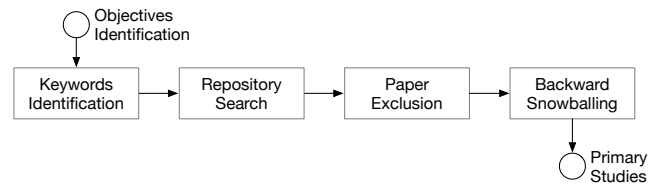


Figure 1: Overview of the literature review process.

**Keywords Identification.** To understand the status quo of incompatible app analyses, we resort to a set of keywords to search for 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.,  $g1 \text{ AND } g2$ , 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\*, deprecate\*, issue\*, evolution*

**Repository Search.** To focus the search, we applied these keywords on all the CORE<sup>1</sup> 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. We then go through these 19 venues one by one and apply 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, IS-SRE, ESEM, ICSME, MSR, ICSCA, SANER, SEAMS, ICST, ISSTA

<sup>1</sup><https://www.core.edu.au/home>

**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 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 4). For example, the work presented by Ki et al. [27], 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. [53], 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. For example, Nielebock et al. [42] 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. [17] 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 9 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 we have selected to search for relevant publications are indeed relevant ones.

Table 2: Full List of Collected and Examined Papers.

Tool/Reference	Year	Venue	Tool availability
ACID[40]	2021	SANER	Available [1]
ACRYL (extension)[48]	2020	EMSE	Open Source [2]
ACRYL[47]	2019	MSR	Open Source [2]
Pivot[57]	2019	ICSE	Available [5]
CiD[35]	2018	ISSTA	Open Source [3]
IctApiFinder[22]	2018	ASE	Open Source [7]
CIDER[24]	2018	ASE	Available [4]
FicFinder (extension)[58]	2018	TSE	Available [6]
FicFinder[56]	2016	ASE	Available [6]

## 2.2 Result

In total, our Systematic Literature Review (SLR) search process identified nine relevant papers (hereinafter referred to as primary



Figure 2: The word cloud of the abstract text in the selected papers.

studies, which are listed in the first column of the 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 some of them are published as executable files on the associated papers' websites. Figure 2 further illustrates the word cloud of the abstract texts among the identified primary publications. Terms such as *Android*, *API*, *compatibility*, *issue*, and *app* remain the most representative ones in the word cloud, suggesting that the collected primary publications are indeed relevant to the topic targeted by this work (hence suitable for our study).

## 2.3 Status Quo Analysis

After identifying the primary publications, we carefully read their full papers to understand how each of their automated compatibility issues detection approaches are 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 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 harvesting. The remaining tools are working as a whole supporting both APIs harvesting and issue detection.

Table 3: Working Process Support of Tools.

Tool/Reference	API Harvest	Issue Detection
ACID[40]	✓	✓
ACRYL (extension)[48]	✓	✓
ACRYL[47]	✓	✓
Pivot[57]	✓	✗
CiD[35]	✓	✓
IctApiFinder[22]	✓	✓
CIDER[24]	✗	✓
FicFinder (extension)[58]	✗	✓
FicFinder[56]	✗	✓

Among the nine primary publications, as shown in Figure 4, after carefully reading their full content, we categorize their compatibility

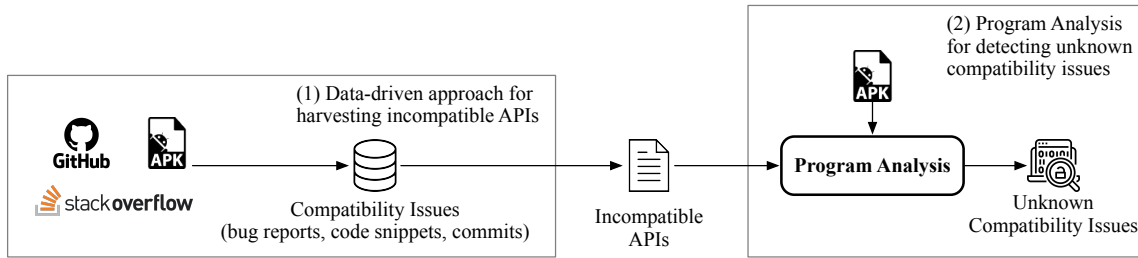


Figure 3: The typical working process of detecting Android compatibility issues.

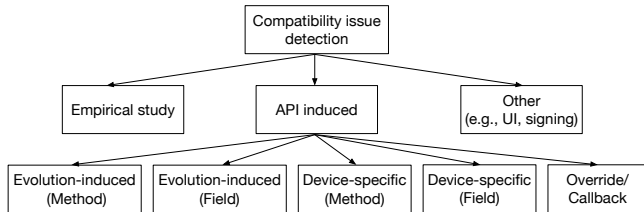


Figure 4: The category of the papers targeting compatibility issues on Android platform.

issue detection capabilities into five types of issues. For each of the considered tools, we further summarize and list its capabilities in Table 4. Columns 2-6 describe the detection of the five different types of compatibility issues in order as described in Figure 4, which are further detailed with concrete examples as follows.

```

1 //Example 1: Evolution-induced(Method)
2 public class MainActivity extends Activity{
3     private TextView mView;
4     protected void onCreate(Bundle bundle) { ...
5 +     if(Build.VERSION.SDK_INT >= 24)
6 +         wrapper(mView, c, s, null, i);
7 +     else
8         mView.startDrag(c, s, null, i);
9     }
10 + private wrapper(View v, ClipData c, ...) {
11 +     v.startDragAndDrop(c, s, o, i);
12 + }
13 }
14 //Example 2: Evolution-induced(Field)
15 public static Bitmap getCacheArt(final Context
    context,final Song song){
16     ...
17     Options options=new Options();
18     options.inDither=false;
19     options.inPreferredConfig=ARGB_8888;
20     ...
21 }
22 //Example 3: Device-specific(Method)
23 Camera mCamera = Camera.open();
24 Camera.Parameters params = mCamera.getParameters();
25 .....
26 + if (android.os.Build.MODEL.equals("Nexus 4") {
27 +     params.setRecordingHint(true);
28 + }
29     .....
30 mCamera.setParameters(params);
31 mCamera.startPreview();
32 //Example 4: Device-specific(Field)
33 private static HttpClient getNewHttpClient() {
34     ...
35 sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_
    HOSTNAME_VERIFIER);

```

```

36 ...
37 }
38 //Example 5: Override/Callback
39 public void onAttach(Context context) {
40     super.onAttach(context);
41     mActivity = (BrowserActivity) context;
42     .....
43 + attachActivity((BrowserActivity) context);
44 }
45 + public void onAttach(Activity activity) {
46 +     super.onAttach(activity);
47 +     if (Build.VERSION.SDK_INT < 23) {
48 +         attachActivity((BrowserActivity) activity);
49 +     }
50 + }
51 + private void attachActivity(BrowserActivity activity
    ) {
52 +     mActivity = activity;
53 +     .....
54 + }

```

Listing 1: Code examples.

**Evolution-induced (Method):** The signatures of some public methods are altered (i.e., removed, newly added, or parameter type changes, etc.) during the evolution of the framework. Example 1 in Listing 1 demonstrates such an example, for which the code snippet is initially reported in [22], where statements beginning with the + signs indicate a possible fix for this incompatibility. The API `startDrag()` called on Line 8 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 [22] and [40], 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 18 of Example 2 in Listing 1. 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 smart-phone manufacturers, some APIs only work on some devices but not on others. Example 3 of Listing 1 demonstrates such an example, originally reported by Wei et al. [57]. Only if the result of the conditional statement for checking the device identifier according

**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 <sup>2</sup>
ACID[40]	✓	✓ <sup>1</sup>	✗	✗	✓	✓	✓
ACRYL (extension)[48]	✓	✗	✗	✗	✗	✗	✓
ACRYL[47]	✓	✗	✗	✗	✗	✗	✓
Pivot[57]	✓	✗	✓	✗	✗	✗	✓
CiD[35]	✓	✗	✗	✗	✗	✓	✓
IctApiFinder[22]	✓	✓ <sup>1</sup>	✗	✗	✗	✓	✓
CIDER[24]	✗	✗	✗	✗	✓	✗	✗
FicFinder (extension)[58]	✓	✗	✓	✗	✗	✗	✗
FicFinder[56]	✓	✗	✓	✗	✗	✗	✗

<sup>1</sup> Only mentioned but not illustrated in detail.

<sup>2</sup> There is no human involvement in the core process, e.g., the learning/knowledge collection phase.

to “Nexus” is true, that is, the corresponding app is indeed running on “Nexus”, the API *setRecordingHint()* on Line 27 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 Example 4 of Listing 1. 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<sup>2</sup> by client Android developers, and their execution will be triggered by the framework. The Example 5 in Listing 1 demonstrates such an example excerpted from [24]. The *onAttach(Context)* callback method at Line 39 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 obviously 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 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 [40], 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. [47] 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. [56, 58] 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.

<sup>2</sup>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.

**RQ1 Findings**

Our literature review reveals several recent approaches to detecting compatibility issues in Android apps aiming at mitigating the impact of fragmentation in the Android community. Although these state-of-the-art approaches are effective in detecting some issues, they all have weaknesses and limitations. 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 many require considerable manual efforts. There is thus a need for new approaches to holistically resolve all the types of compatibility issues, and these approaches should be systematic and automatic, accounting for all possible issues in the ecosystem and newly emerging issues.

**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 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 [22] 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** [35] first models the lifecycle of 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** [22] 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 behaviors on different OS versions. More importantly, they find that the additional supporting code share 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 inter-procedural control flow graph (ICFG) by Soot for Android Apps and then extracting Android APIs 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** [24] 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) [60] 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 [52].

**FicFinder** [58] 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 a context-free grammar. Therefore, the algorithm identifies the issue-inducing Android APIs as well as their dependencies, analyses the calling context, and then compares with the modeled issue-triggering context. To analyze the dependencies issue-inducing API related, the algorithm carries out an inter-procedural backward slicing on callsite to acquire the slices of statements on the basis of program dependence graph [20]. If the triggering context is not considered before invoking the API, a new issue is reported. To implement this artefact, the well-known static analysis framework, Soot [52], is utilized.

Each of the selected tools requires specific configuration. As the detection result relies on these basic configure 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<sup>3</sup> 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) 20 Android apps for CIDER, seven apps for CiD, eight apps for IctApiFinder, 30 for FicFinder.<sup>4</sup> 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 should not impact the results of the replicability study.

### 3.3 Result

When we do our replication, CIDER and CiD do have exactly the same outputs on the original Android Apps while FicFinder and IctApiFinder have some different outputs regards their original experimental Apps. We now detail the 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 was 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. The one is compatibility issues in TP and FP, and the other is Good Practice (GP) meaning already fixed issues. After we reproduce in our local environment, seven of them do have different output compared with the original ones presented in the 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

<sup>3</sup>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.

<sup>4</sup>The FicFinder authors have actually considered 53 Android projects but only 30 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.

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] was revealed 6 GP but we detected with only 3 GP, MozStambler [13] contained 1 issue and one GP but we only detected with 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-like 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 slightly updates of the tools or the evaluated apps.

## 4 COMPARISON STUDY (RQ3)

The last research question aims to empirically compare the state-of-the-art tools targeting the detection of compatibility issues in Android apps. 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 resort to the following two datasets to support the comparison study.

- **Dataset1:** The same 65 apps used for the replicability study as discussed in Section 3.
- **Dataset2:** 645 Android apps selected from the AndroidCompass dataset [42]. AndroidCompass contains a dataset of git

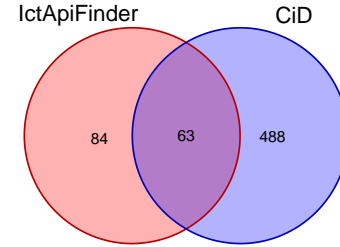
**Table 5: Experimental results obtained based on the 65 apps located in Dataset1.**

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
libtorrent	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
materialbook	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

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 (i.e., the apps compiled based on their immediate previous commit). We could further collect the actual compatibility issues based on the compatibility checks added in the fix commits for each of the compiled apps. This study will leverage this information as partial ground truth to support the comparison study. Unfortunately, several app projects are no longer available on Github, while some others cannot be easily compiled into APKs (e.g., due to missing library dependencies), we have to exclude them. Eventually, we were able to collect 645 apps to fulfill this dataset.

### 4.3 Result

**Results on Dataset1.** We first launch 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 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.

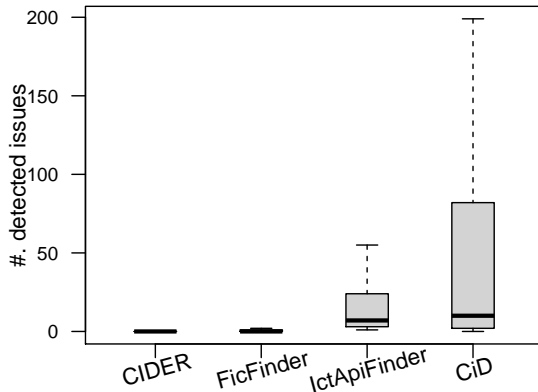
**Figure 5: Venn diagram of incompatible APIs utilized in IctApiFinder and CiD.**

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, there are only 8 apps being 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). 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.

While both IctApiFinder and CiD yield significantly more results than FicFinder and they do take systematic approaches to collect incompatible APIs, their results are quite different. Among the 28 apps successfully analyzed by both of these two tools, IctApiFinder and CiD respectively yields in total 766 and 1,017 issues, for which only 52 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. The 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 time framework of Android 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.

**Results on Dataset2.** We then launch the selected tools on Dataset2, which contains a large number of real-world Android apps selected from the AndroidCompass compatibility checks dataset.



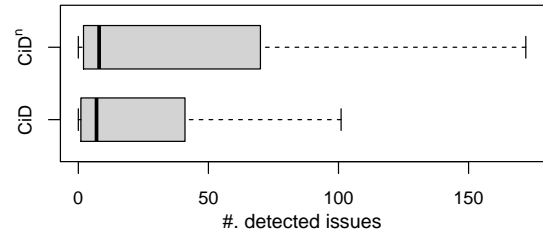


**Figure 6: Compatibility issues detected by different detection tools against Dataset2.**

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<sup>5</sup> at 0.001.

Observant readers may have noticed 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 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.

<sup>5</sup>Given a significance level  $\alpha = 0.001$ , if  $p\text{-value} < \alpha$ , there is one chance in a thousand that the difference between the datasets is due to a coincidence.



**Figure 7: Comparison between original CiD and API life-cycle extended CiD.**

### RQ3 Findings

Comparing the selected four tools on the same datasets, CiD is able to yield more compatibility issues than the other tools, followed by IctApiFinder, and then FicFinder. Their results are however not well overlapped, suggesting the existing tools are complementary to each other and yet still have limitations to achieve sound 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 detection of compatibility issues.

## 5 DISCUSSION

We 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.

### 5.1 Implication

**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 the new system releases into account. Besides, many device vendors always release lots of different models as their own publish step. To detect the newly introduced compatibility issues, these tools need to be refined once new version system released and 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<sup>n</sup>. We then launch CiD<sup>n</sup> to analyze the apps in Dataset2. Figure 7 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.

**Extracting Complete Incompatible APIs:** The state-of-the-art tools all heavily rely on summarized incompatible APIs regardless of whether they are manually summarized or systematically collected. However, tools we analyzed all have their own approach to collect incompatible APIs, which not only sets obstacles to have a fair comparison but also undermines capacities of the detection tools. Therefore, an effective approach to extract a complete list of incompatible APIs is necessary.

**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 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. [38], 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 to fix them. 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.

## 5.2 Threats to validity

As it is the chase of most empirical studies, there are threats to validity associated with the results we presented. 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 an input parameter. However, the version of the Soot and Android framework may be still different because we cannot know the exact versions leveraged in every detection tool. To mitigate this threat, we meticulously align the configurations among them as much as we can to provide an approximately 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 own 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 the future, we will try to align the extraction approach to achieve a fair comparison.

## 6 RELATED WORK

In recent years, compatibility issues have been a hot topic in the Android community [19, 25, 29, 45, 47, 53, 62]. 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. [36, 37] 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 [32] and inconsistent release time of Android apps [31]. Scalabrino et al. [47] 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 [48] 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. [59] 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. [41] 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. [40] 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 [34]. Qiu et al. [43] did an extensive comparison among three most prominent static analysis tools including

FlowDroid [14] combined with IccTA [30], DroidRA [33, 51], AmanDroid [54, 55], and DroidSafe [21]. 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 [16, 23, 44, 46, 63]. Sawant et al. [46] 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. [18] 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. [50] 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.

## 7 CONCLUSION

In this paper, we have conducted a literature review on research works 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 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.

## ACKNOWLEDGEMENTS

This work is supported by ARC Laureate Fellowship FL190100035, ARC 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] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oetean, 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.
- [15] 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.
- [16] 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.
- [17] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A Large-Scale Study of Application Incompatibilities in Android. In *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*.
- [18] 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.
- [19] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael O'Boyle. 2020. M3: Semantic API Migrations. *arXiv preprint arXiv:2008.12118* (2020).
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Citeseer.
- [27] 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.
- [28] 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).
- [29] 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.
- [30] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Oetean, 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.
- [31] 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)*.
- [32] 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)*.
- [33] Li Li, Tegawendé F Bissyandé, Damien Oetean, 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)*.
- [34] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Oetean, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).
- [35] 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.

- [36] Li Li, Jun Gao, Tegawendé F Bisseyandé, 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.
- [37] Li Li, Jun Gao, Tegawendé F Bisseyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. Cda: Characterising deprecated android apis. *Empirical Software Engineering* (2020), 1–41.
- [38] 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)*.
- [39] 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).
- [40] 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.
- [41] Marco Mobilio, Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. 2020. 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.
- [42] 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).
- [43] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: Flow-droid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–186.
- [44] 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.
- [45] 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.
- [46] 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.
- [47] 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.
- [48] 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.
- [49] 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).
- [50] 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.
- [51] Xiaoyu Sun, Li Li, Tegawendé F Bisseyandé, 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).
- [52] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASON First Decade High Impact Papers*. 214–224.
- [53] 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.
- [54] 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.
- [55] 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.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] 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 42nd International Conference on Software Engineering (ICSE)*. IEEE, 886–898.
- [60] 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.
- [61] 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).
- [62] 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)*.
- [63] 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.