Understanding and Detecting Callback Compatibility Issues for Android Applications

Huaxun Huang
The Hong Kong University of Science and Technology
Hong Kong, China
hhuangas@cse.ust.hk

Yepang Liu*
Southern University of Science and Technology
Shenzhen, China
liuyp1@sustc.edu.cn

Lili Wei*
The Hong Kong University of Science and Technology
Hong Kong, China
lweiae@cse.ust.hk

Shing-Chi Cheung*
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

ABSTRACT

The control flows of Android apps are largely driven by the protocols that govern how callback APIs are invoked in response to various events. When these callback APIs evolve along with the Android framework, the changes in their invocation protocols can induce unexpected control flows to existing Android apps, causing various compatibility issues. We refer to these issues as *callback compatibility issues*. While Android framework updates have received due attention, little is known about their impacts on app control flows and the callback compatibility issues thus induced. To bridge the gap, we examined Android documentations and conducted an empirical study on 100 real-world callback compatibility issues to investigate how these issues were induced by callback API evolutions. Based on our empirical findings, we propose a graph-based model to capture the control flow inconsistencies caused by API evolutions and devise a static analysis technique, CIDER, to detect callback compatibility issues. Our evaluation of CIDER on 20 popular open-source Android apps shows that CIDER is effective. It detected 13 new callback compatibility issues in these apps, among which 12 issues were confirmed and 9 issues were fixed.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; • **Human-centered computing** → **Smartphones**; • **General and reference** → **Empirical studies**;

KEYWORDS

Android API, empirical study, static analysis, callback compatibility

---

*Shing-Chi Cheung and Yepang Liu are the corresponding authors. Lili Wei is also a visiting student at the Southern University of Science and Technology.

---

1 INTRODUCTION

Android apps are event-driven. *Callback APIs*, overridden and implemented in Android apps, are invoked by the underlying operating system in response to various events. The control flows of Android apps are largely driven by the callback API invocation protocols, which define how callback APIs are invoked to handle events. However, callback APIs are fast evolving along with the Android system [47]. Such evolutions can change callback API invocation protocols and cause apps to exhibit inconsistent control flows when running on different versions of Android system. These inconsistencies make it difficult for app developers to correctly maintain program dependencies among different overridden callback APIs across different API levels. As a result, various compatibility issues arose. In this paper, we refer to such issues induced by callback API evolutions as *callback compatibility issues*.

Several techniques have been devised to help analyze the control flows among callback APIs for Android apps. For example, FlowDroid [34] can construct dummy main methods to emulate the execution orders of lifecycle callback APIs for activity components of Android apps. Barros et al. [35] also presented a technique to detect implicit control flows in Android programs. While these techniques help recover control flows for Android apps, none of them can identify inconsistent app control flows caused by the callback API evolutions. Researchers have also studied the evolutions of Android APIs. Linares et al. [46] analyzed how change-proneness and error-proneness of Android APIs affect the ratings of Android apps. McDonnell et al. [47] analyzed the historical data of a set of Android apps to understand the relationship between API evolutions and API adoption. Wei et al. [52] conducted the first longitudinal study on the evolutions of permissions in the Android ecosystem. Wei et al. [51] investigated the fragmentation-induced compatibility issues in Android apps including those that are induced by Android API evolutions. While these pieces of work studied the evolutions of Android APIs, none of them considered the changes in callback

API invocation protocols introduced by these evolutions and the control flow inconsistencies thus induced to Android apps.

In this paper, we first study the evolutions of callback APIs to see how they can affect the invocation protocols of callback APIs. We characterize the callback compatibility issues induced by such the evolutions. To this end, we conducted an empirical study based on the API Reference in Android Developers [6], Android API Differences Reports [2], Android Open Source Project [5], and 100 callback compatibility issues from 50 open-source Android apps. Our study aims to answer the following two research questions:

- **RQ1 (Callback API Evolutions and Invocation Protocols)** *How do the evolutions of callback APIs change the API invocation protocols? Are there any frequently-evolved callback APIs?*

- **RQ2 (Causes of Callback Compatibility Issues)** *How do the evolutions of callback APIs induce callback compatibility issues? Are there major issue-inducing causes?*

In answering RQ1, we identified two common types of callback API evolutions: *API reachability change* and *API behavioral modification*. The former type changes the reachability of callback APIs while the latter type changes the invocation conditions and behaviors of the concerned callback APIs. Both of the two types of changes in callback API evolutions can alter app control flows and induce callback compatibility issues. In answering RQ2, we observed that callback compatibility issues commonly arose if apps fail to properly handle inconsistent control flows induced by the callback API evolutions. This motivates us to propose a graph-based model to capture such inconsistent app control flows when running on different API levels for the analysis of callback compatibility issues.

Based on these observations, we propose CIDER, a **C**allback Compatibility **I**ssue **De**tecto**r** based on static analysis. It leverages our proposed graph-based model to generate callback control flow graphs capturing inconsistent control flows of callback APIs in Android apps. To evaluate the performance of CIDER, we applied it to 20 real-world open-source Android projects collected from GitHub [15]. CIDER successfully detected 13 previously-unknown callback compatibility issues, achieving 92.9% precision. It significantly outperformed Lint [4], a widely-used static analyzer for Android apps, by detecting more callback compatibility issues while generating much fewer false positives. We reported our detected issues to the original developers of the corresponding Android apps for their feedback. Among the 13 issues, 12 were confirmed by the developers and nine of them were fixed shortly afterwards. This shows that CIDER can precisely detect callback compatibility issues of interest to the Android app developers. In summary, this paper makes the following contributions:

- We conducted an empirical study to understand the callback API evolutions in the Android official documents and framework code, and further characterized their impact on app control flows. We also investigated the changes in callback API invocation protocols and identified major causes of callback compatibility issues. Our dataset is available at [10].

- We formulated a graph-based model to capture app control flow inconsistencies induced by callback API evolutions. Based on this model, we proposed a static analysis technique CIDER to detect callback compatibility issues in Android apps.
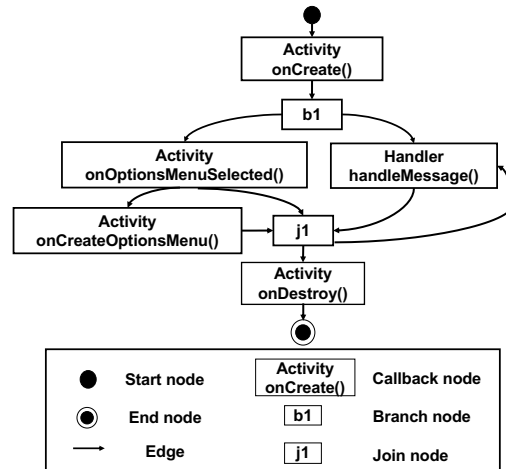

Figure 1: A CCFG with five callback APIs.

- We evaluated CIDER on 20 open-source Android apps. CIDER outperformed a widely-used static analyzer, Lint, and successfully detected 13 previously-unknown callback compatibility issues, 12 of which were confirmed or fixed by the app developers.

2 BACKGROUND

Callback APIs and callback control flow graph. Android apps are event-driven. Callback APIs (see Figure 1 for examples) encapsulate code that should be executed to handle various system and user events. Due to the event-driven paradigm, control flows of an Android app are mainly determined by flow of triggering events. In other words, an Android app's execution is largely driven by control flows across callback APIs.

In this paper, we leverage the concept of *Callback Control Flow Graph (CCFG)* to capture the control flows between callback APIs in Android apps and discuss the issues induced by the evolution of the callback APIs. CCFG was introduced by Yang et al. [54] to aid control flow analysis for Android apps. Figure 1 shows an example of a simplified CCFG. A CCFG is a directed graph that consists of two types of nodes: *callback node* and *helper node*. Callback nodes represent the invocations of callback APIs while helper nodes represent branch and join points of control flows. As shown in Figure 1, there are five callback nodes in the CCFG, each of which is labeled with the name of the corresponding callback API and the concerned library class. Node b1 and j1 are two helper nodes indicating that the subsequent callback APIs of the *branch node* b1 can be executed in any order until the *join node* j1 is reached. Edges between the nodes in a CCFG represent the transfer of control flows between the nodes (i.e., the order of execution). For example, the edge between the callback nodes onOptionsMenuSelected and onCreateOptionsMenu indicates that the onCreateOptionsMenu callback will be invoked after the onOptionsMenuSelected callback completes its execution.

Android API evolutions and callback compatibility issues. The Android framework is fast evolving. By 2017, there were already 27 different Android API levels [3]. Each API level introduces API

```
01.    public void onAttach(Context context) {
02.        super.onAttach(context);
03. -      mActivity = (BrowserActivity) context;
04. -      ................
05. +      attachActivity((BrowserActivity) context);
06.    }
07. +  public void onAttach(Activity activity) {
08. +      super.onAttach(activity);
09. +      if (Build.VERSION.SDK_INT < Build.VERSION_CODES.M) {
10. +          attachActivity((BrowserActivity) activity);
11. +      }
12. +  }
13. +  private void attachActivity(BrowserActivity activity) {
14. +      mActivity = activity;
15. +      ................
16. +  }
17.    public void onActivityCreated(Bundle savedInstanceState) {
18.        mActivity.set(this);
19.        ................
20.    }
```

Figure 2: The patch for WordPress issue 6906 [33].

Figure 3: The CCFG for the code snippet in Figure 2.

updates for various purposes such as adding new features, bug fixing, performance optimization and code restructuring. These updates can induce compatibility issues in Android apps [51].

While non-callback APIs are implemented in the Android framework and can be directly invoked by Android apps, callback APIs are invoked by the Android system in response to occurred events. Callback APIs are often overridden and customized by developers to implement app-specific functionalities. As the Android framework evolves, the callback APIs and their invocation protocols also evolve. Since the invocation protocols of callback APIs affect app control flows, the evolutions of callback APIs and changes in their invocation protocols can have significant influence on Android apps' runtime behavior. Various compatibility issues would arise if such changes, which might be subtle, are not timely identified and properly handled. This is particularly the case if an app contains certain code that has control/data dependencies on multiple callback APIs. This motivates us to study Android callback API evolutions and their induced compatibility issues, which are referred to as *callback compatibility issues* hereafter.

3   MOTIVATING EXAMPLE

This section presents an example to illustrate how callback API evolutions can induce callback compatibility issues and discusses the challenges of the issue detection. The example is extracted from issue 6906 in WordPress [33], a popular website and blog builder with more than 5 million downloads. According to the issue report, WordPress would crash due to a NullPointerException (NPE for short) caused by the evolution of a lifecycle callback API in a Fragment, which is an Android user-interface component with its own lifecycle [14]. In Android apps, a Fragment's lifecycle callback APIs need to be carefully implemented to respond to the events that would cause the Fragment to change its lifecycle stage. The crash was induced by a new lifecycle callback onAttach(Context) of Fragment, which was introduced at Android API level 23. The callback can be implemented to handle the user events that cause a
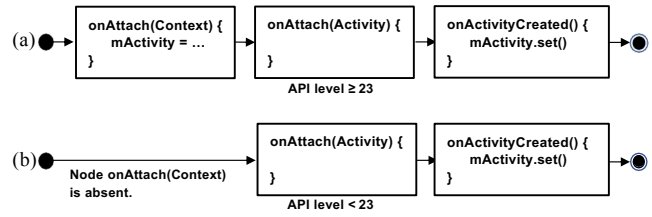
Fragment to be attached to a parent component. We explain how WordPress issue 6906 arose due to the callback API evolution below.

Figure 2 gives the relevant code snippet of the crashing issue and the patch written by the developers. In line 3, the variable mActivity is initialized in the callback onAttach(Context). The variable is later used in another callback onActivityCreated (line 18). Figure 3 shows the CCFG for the code snippet in Figure 2. The edges in Figure 3(a) indicate the control flows between the concerned Fragment callback APIs at API level 23 or higher. According to the CCFG, the following three callback APIs will be invoked sequentially: onAttach(Context), onAttach(Activity) and onActivityCreated. However, the callback API invocation sequence differs when the API level is lower than 23, where onAttach(Context) will be skipped and only onAttach(Activity) and onActivityCreated will be invoked as shown in Figure 3(b). This is because the callback API onAttach(Context) was first introduced at API level 23 and thus unavailable at API levels lower than 23. When WordPress runs on a device with an API level lower than 23, onAttach(Context) will not be invoked and the variable mActivity (its default value is null) will not be initialized. As a result, the invocation of onActivityCreated will lead to an NPE. To fix this issue, WordPress developers revised the code of onAttach(Activity) to properly initialize mActivity if the runtime API level is lower than 23 (lines 9–11).

The above issue arose due to a CCFG inconsistency induced by the evolution of the callback API onAttach(Context) at API level 23. The inconsistency affects the def-use chain of the variable mActivity. Since inter-callback data flows are common in Android apps [38], such callback API evolutions can easily induce compatibility issues. To detect such issues, we not only need to have a deep understanding of CCFG inconsistencies caused by callback API evolutions but also need to analyze the dependencies between app callback APIs. To the best of our knowledge, none of the existing studies on Android compatibility issues investigated the impact of callback API evolutions on app control/data flows and inter-callback dependencies [41, 48, 51]. To bridge the gap, we conducted an empirical study to understand the callback compatibility issues induced by callback API evolutions in Android. The study results further guided us to design a technique to automatically detect the issues.

4   EMPIRICAL STUDY METHODOLOGY

4.1   Study Setup for Answering RQ1

To answer RQ1, we conduct an empirical study to understand the evolutions of Android callback APIs and the changes in callback API invocation protocols. The empirical data come from the API

Table 1: Keywords used in RQ2.

compatibility	compatible	compat	deprecated
deprecation	honeycomb	kitkat	jellybean
icecreamsandwich	lollipop	marshmallow	nougat
gingerbread	oreo	api	`Build.VERSION`

Reference in Android Developers [6], Android API Differences Reports [2], and the Android Open Source Project [5]. In order to have a comprehensive understanding of the callback API evolutions, we chose to analyze the Android framework from API level 1. We describe the process in the following.

We first need to locate a list of callback APIs from the API Reference since it does not provide a list of callback APIs. To achieve this, we study the specifications of well-known callback APIs triggered by the lifecycle events of `Activity`, `Fragment` and `Service`, which are the key components of Android apps. We identified the common characteristics of these API specifications and formulated our search criteria to locate callback APIs. Specifically, we collected three kinds of APIs: (1) the APIs whose name starts with "on", (2) the APIs whose description includes keywords "callback" or "listener", (3) the APIs whose description contain sentences with at least one word from each of the following two sets of keywords (case insensitive): {"*called*", "*invoked*", "*notified*"}, and {"*when*", "*before*", "*after*"}. We treated these APIs as callback API candidates and extracted their information from API Reference. As a result, we identified 5,589 callback API candidates out of 42,591 Android framework APIs. Then, two of the authors cross-examined these 5,589 API candidates to further filter out noises in the set of candidates and kept only callback APIs for the empirical study. Eventually, we collected 2,589 callback APIs to form our callback API list (denoted as $\mathcal{L}_{API}$).

We further identify the evolutions of the callback APIs in $\mathcal{L}_{API}$. However, simply analyzing the API Reference is inadequate because such a constantly-evolving document can be incomplete (e.g., they do not provide information for certain removed APIs). To obtain more complete information of Android callback APIs, we further analyzed Android API Differences Reports, which provide the full history of Android framework API updates. For each callback API update, we then studied its associated code commits in the Android Open Source Project (AOSP for short) [5]. Specifically, we studied the code changes, related comments and commit logs to understand how each update affects the callback API invocation protocol.

4.2 Study Setup for Answering RQ2

In RQ2, we aim to understand how callback API evolutions can affect the software quality of Android apps and characterize the induced callback compatibility issues. To achieve this, we need to construct a dataset of callback compatibility issues from real-world Android apps. For data collection, we searched two popular open-source app hosting platforms: F-Droid [12] and GitHub [15]. We collected the Android apps that (1) have at least one commit after October 2017 (actively-maintained), (2) have received at least 50 stars (popular), and (3) have a public issue tracking system (traceable). This resulted in a set of 275 apps. For these 275 candidate apps, we further searched for patched callback compatibility issues by mining their code repository. Specifically, we searched for commit

logs, issue reports, and code diffs that contain the keywords listed in Table 1. These keywords include the names of major API levels, the name of the commonly-used class to check the runtime API level (`Build.VERSION`) and several general terms related to compatibility (e.g., we observed that developers often use "api" to represent "API level" when discussing issues). To ensure the usefulness of the dataset, we only collected callback compatibility issues that can occur on devices with an API level 10 or above (older API versions have no market share [3]). As a result, we obtained 100 real callback compatibility issues from 50 out of the 275 apps as our empirical study dataset. For each collected issue, we carefully identified the evolved callback APIs and studied how the callback API evolutions affected the concerned app's CCFG and induced callback compatibility issues. We also analyzed the corresponding issue reports and issue-fixing code commits, which contain patches, to understand the practices of Android developers to fix callback compatibility issues.

5 EMPIRICAL STUDY RESULTS

5.1 RQ1: Callback API Evolutions and Invocation Protocols

As discussed earlier, to answer RQ1, we identified the updates for each callback API in $\mathcal{L}_{API}$. In the resulting dataset, we observed four ways of callback API evolutions: *API introduction*, *API deprecation*, *API behavioral modification* and *API removal*. We further examined the code commits in AOSP that introduced the updates of callback APIs to study the common changes in the APIs' invocation protocols. We present our findings in this section.

5.1.1 Callback API Invocation Protocol Change. There are two major categories of changes to the callback APIs' invocation protocols due to callback API evolutions.

Reachability change. In most cases, a callback API's reachability is changed when it evolves, making it accessible only at specific Android API levels. For example, the onAttach(Context) callback API defined in the `Fragment` class was introduced at API level 23. Therefore, the callbacks that override it cannot be invoked at runtime when the API level is below 23. There are 1,762 such evolved callback APIs. The evolutions alter the invocation protocols of the concerned callback APIs by enabling or disabling the APIs' invocations depending on the runtime API levels. This can significantly affect the control flow of Android apps. In Section 5.2, we will show that the reachability changes of callback APIs can commonly induce callback compatibility issues in Android apps.

API behavioral modification. There are 37 callback APIs whose behaviors were updated. The updates did not add, remove or change the reachability of callback APIs, but led to API behavioral modification. In our dataset, we observed three common types of such evolutions: (1) API declaration changes (e.g., the callback API onNotificationPosted's modifier abstract was removed in API level 21), (2) calling condition or order changes (e.g., since API level 11, if the API invalidateOptionsMenu, which declares that the options menu has changed, is called, the callback API onCreateOptionsMenu will also be called by the system when the options menu needs to be displayed), and (3) the values passed to method parameters change (e.g., the second parameter of the callback API
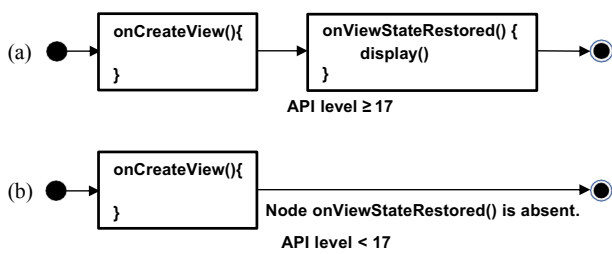
Figure 4: CCFG node inconsistency example.

Figure 5: CCFG edge inconsistency example.

onNewPicture(WebView, Picture) will always receive a null reference since API level 18). Such behavioral modifications will cause specific changes to the APIs' invocation protocols since the concerned API may undergo different behavioral changes. As a result, the callback compatibility issues induced by such callback API evolutions are also specific to the concerned callback APIs. We will further discuss these callback compatibility issues in Section 5.2.

5.1.2 Commonly-Evolved Callback APIs. We analyzed the callback APIs that have been updated since their first introduction and observed that most of them are defined in four classes: `Activity`, `Fragment`, `Service` and `WebView`. These classes are commonly-used by Android apps for UI displays (`Activity` and `Fragment`), background tasks (`Service`), and web browsing (`WebView`). Their callback APIs account for 25% of the callback APIs in $\mathcal{L}_{API}$. Around 55% of all API updates in our dataset are related to the callback APIs in these four classes. Frequent updates of such commonly-used callback APIs can easily cause callback compatibility issues that would affect a large number of Android apps. This motivates the need of automated tools to support callback compatibility issue detection. In the next section, we will discuss how callback API evolutions can induce callback compatibility issues by changing app control flows. With this knowledge, we then propose a technique for callback compatibility issue detection.

Answer to RQ1: Callback API evolutions can modify API invocation protocols, changing the reachability or functionality of the evolved APIs. Such modifications affect app control flows. Frequent updates of callback APIs in a few widely-used classes commonly induce callback compatibility issues.

5.2   RQ2: Causes of Callback Compatibility Issues

In RQ2, we study how the callback API evolutions observed in RQ1 can change app CCFGs and induce callback compatibility issues. We investigated the 100 callback compatibility issues collected from open-source projects on GitHub (Section 4.2). We found that all these 100 issues arose from two types of CCFG inconsistencies induced by callback API evolutions.

5.2.1 CCFG structural inconsistency. A vast majority of the issues (89 out of 100) arose from the CCFG structural inconsistencies across API levels. CCFG structural inconsistencies occur when the set of nodes or edges of an app's CCFG changes across different API
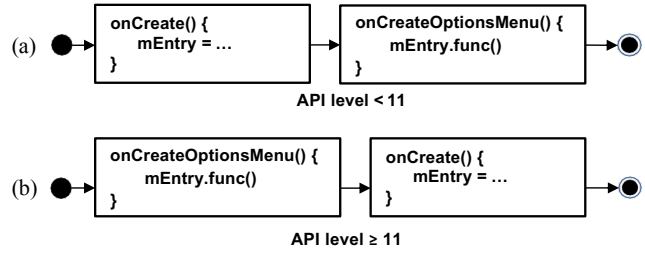
levels. From the 89 issues, we found that the callback compatibility issues arising from node inconsistencies and those arising from edge inconsistencies differ in nature. In addition, the issues arising from node inconsistencies are most common.

CCFG node inconsistency. There are 82 issues induced by CCFG nodes that exist in some API levels but disappear in other API levels. This is commonly caused by API reachability changes (Section 5.1.1). The introduction of a callback API at an API level adds a new node to concerned CCFGs, while the removal of an API deletes a node from the CCFGs. In both cases, the node can only be found in the CCFGs for certain API levels. This would lead to inconsistent app behaviors across different API levels, causing callback compatibility issues.

With further examination, we found that the absence of a CCFG node can (1) destruct variable def-use chains or (2) eliminate the invocations of key APIs that can cause user-perceivable app behaviors. Our motivating example in Section 3 gives an instance of such inconsistencies. In the example, the onAttach(Context) node is absent from the app's CCFG for API levels lower than 23. The absence of the node destructs the def-use chain of the variable mActivity, causing the app to crash. The commit 2681e87 in Bitmask [7] filed another issue induced by invoking a key API in a callback node that is absent in CCFGs for specific API levels. Figure 4 shows the inconsistent CCFGs. As the figure shows, the API display is invoked in the callback API onViewStateRestored to display information on the screen. Since this callback API is not available until API level 17, the API display will not be called when Bitmask runs on devices with an API level lower than 17. As a result, the information cannot be properly displayed. The root cause of this issue is that some APIs that can cause perceivable changes at runtime (e.g., display) are not invoked because of CCFG node absence. This motivates us to maintain a list of such APIs to support the automated detection of callback compatibility issues via static analysis (Section 6.2).

CCFG edge inconsistency. The remaining 7 of the 89 issues were induced by CCFG structural inconsistencies that occurred when the set of CCFG edges varied across different API levels while the set of CCFG nodes remained the same. Such edge inconsistencies can also destruct variable def-use chains and cause callback compatibility issues by scrambling the execution order of callback APIs. For example, the commit 467d6e8 of KeePassDroid [16] documented an issue caused by CCFG edge inconsistency. Figure 5 shows the concerned CCFGs. Before API level 11, the Android system will only trigger the callback API onCreateOptionsMenu each

time when the users open the options menu and before that the on-
Create callback of the menu's enclosing UI component must have
been invoked. However, at API level 11, a new API `invalidateOp-`
`tionsMenu` was introduced for developers to force the invocation
of `onCreateOptionsMenu`. This makes it possible to invoke `onCre-`
`ateOptionsMenu` before `onCreate`. In this case, the use statement
of `mEntry` in `onCreateOptionsMenu` can be executed before the
def statement of `mEntry` in `onCreate`, causing the app to crash.

5.2.2 CCFG non-structural inconsistency. The remaining 11 of the
100 issues were caused by CCFG non-structural inconsistency. In
these cases, the sets of nodes and edges in CCFGs stayed the same
but the behaviors of the callback APIs were inconsistent across
different API levels. The non-structural inconsistencies were in-
duced by the behavioral modifications of callback APIs such as API
modifier changes (from abstract to non-abstract), API triggering
condition changes and so on. For instance, developers of Omni-
Notes encountered an issue of this type (issue 248 [21]). The issue
was induced by a change in the modifier of the callback API on-
`NotificationPosted`. At API level 21, the API is changed from
abstract to non-abstract. When the API is overridden, the subclass
version invokes the superclass version. This works well on devices
with an API level 21 or above (because the superclass version of the
API is a concrete method). However, when the app runs on devices
with an API level lower than 21, it would crash since the superclass
version of the API is an abstract method that cannot be invoked.

To summarize, callback API evolutions can induce various CCFG
inconsistencies. Such inconsistencies can affect app runtime behav-
iors and cause callback compatibility issues. Among our studied
issues, a majority of them were caused by structural inconsistencies
of CCFGs. As discussed in Section 5.2.1, CCFG structural incon-
sistencies often destruct variable def-use chains or eliminate the
invocations of key APIs that can cause user-perceivable app be-
haviors. It should be noted that the occurrence of these callback
compatibility issues heavily depends on how developers override
and implement the callback APIs (we observed that definitions and
usages of variables and API invocations commonly occur in the
method body of the overridden callback APIs in Android programs).
This motivates us to combine CCFG inconsistencies and the analysis
of the callback code to automatically detect callback compatibility
issues for Android apps (Section 6).

> **Answer to RQ2:** Callback compatibility issues are commonly
> induced by two types of app CCFG inconsistencies arising from
> callback API evolutions. Among them, CCFG structural incon-
> sistencies, which could affect app control flows, caused the
> majority of our studied issues.

6 AUTOMATED DETECTION OF CALLBACK COMPATIBILITY ISSUES

In Section 5.2, we made a key observation that *callback compatibil-
ity issues are commonly induced by CCFG structural inconsistencies
at different API levels.* The structural inconsistencies can destruct
variable def-use chains or eliminate the invocations of key APIs
that can cause user-perceivable changes (e.g., UI display APIs). This

motivates us to construct an app's CCFG at each API level to de-
tect potential callback compatibility issues. However, analyzing the
control flow inconsistencies induced by callback API evolutions is
challenging because precisely constructing an app's CCFG at an API
level is non-trivial as an app component can implement many call-
backs and the timing of their invocations can be non-deterministic.
To address this challenge, we first propose a graph-based model,
Callback Invocation **P**rotocol **I**nconsistency **Graph** (PI-GRAPH), to
capture the structural invocation protocol inconsistencies occur-
ring in an app across API levels (Section 6.1). Then we propose a
technique CIDER, which leverages the PI-GRAPH model, to detect
callback compatibility issues caused by CCFG structural inconsis-
tencies (Section 6.2). For issues induced by CCFG non-structural
inconsistencies, which are much less common, we leave them to
our future work.

6.1 The PI-GRAPH Model

A PI-GRAPH is a directed graph that models the inconsistencies in
the callback API invocation protocols across multiple API levels.
A PI-GRAPH is created based on different versions of the Android
software platform. There are two categories of nodes in a PI-GRAPH:
(1) callback node N_c denoting a callback API, and (2) two types of
helper nodes N_{pre} and N_{suc} denoting the preceding and succeeding
point of PI-GRAPH, respectively. An edge $E = \langle n_s, n_e, APILevel \rangle$
indicates the execution order of two callback nodes n_e and n_s: n_e
will be executed after n_s if the runtime API level is within the
interval $APILevel$, where $APILevel$ is in the form $[a, b]$, in which a
and b specify the lowest and highest API level, respectively. A PI-
GRAPH only contains the nodes for those callback APIs that reside
in the same class as the evolved callback API.

Figure 6(a) shows an PI-GRAPH example. In this example, the
PI-GRAPH captures the CCFG structural inconsistencies induced by
the evolution of the callback API `onAttach` invoked by WordPress
across multiple API levels as discussed earlier in Section 3. Besides
the preceding node N_{pre} and the succeeding node N_{suc}, there are
two callback nodes representing two corresponding callback APIs.
Four edges in the graph indicate the different execution orders of
the callback APIs for different API level intervals. For example, Edge
② in the figure indicates that `onAttach(Activity)` can be exe-
cuted after the execution of `onAttach(Context)` when API level is
within [23, 27]. Based on the PI-GRAPH model, we can generate sim-
plified CCFGs for Android apps by traversing the PI-GRAPH along
edges that are labelled with different API level intervals. The gen-
erated CCFGs can then facilitate control flow analysis for Android
apps with regard to different API levels. In the next section, we
will discuss how PI-GRAPH can be used to generate app CCFGs and
detect callback compatibility issues.

To demonstrate the usefulness of the PI-GRAPH model, we man-
ually derived seven PI-GRAPHs by analyzing API Reference and the
framework source code. These seven PI-GRAPHs concern different
callback APIs in the classes `Activity`, `Fragment`, and `WebView-`
`Client`. According to our empirical study, the evolutions of these
callback APIs commonly caused callback compatibility issues on
recent Android versions.

Algorithm 1: Callback Compatibility Issue Detection.

Input : An Android apk file *apk*;
           a list of PI-Graphs Π;
           a list of key APIs Λ.

Output: Detected compatibility issues

1 *apk.apiLevels* ← GetAPILevels(*apk.config*)
2 **foreach** *π in* Π **do**
3   *apk.classList* ← GetClassesFromPIGraph(*π.apiList*)
4   **foreach** *c in apk.classList* **do**
5   Γ ← GenCCFGFromPIGraph(*π, c, apk.apiLevels*)
6   **foreach** *ccfg in* Γ **do**
7   **if** *CheckUseWithoutDef(ccfg)* **then**
8   report a callback compatibility issue.
9   **if** *CheckMissingKeyAPIs(*Γ, Λ*)* **then**
10   report a callback compatibility issue.

6.2 The Cider Approach

Cider leverages PI-Graphs to detect callback compatibility issues in Android apps by constructing simplified app CCFGs for different API levels. The CCFGs integrate app code into predefined PI-Graphs. Cider then performs static analysis on the CCFGs to detect callback compatibility issues.

Algorithm 1 describes the running process of Cider. It takes an Android app (in apk format), a list of predefined PI-Graphs (denoted as Π), and a list of key APIs (denoted as Λ) as input. Cider first checks the configuration file of the app under analysis to obtain its supported API levels (GetAPILevels in line 1). For each PI-Graph *π* in Π, Cider first identifies all app classes that extend the class that defines the callback APIs in *π* by invoking GetClassesFromPIGraph (line 3). For each class *c* going to be checked, Cider generates a set of simplified app CCFGs (denoted as Γ) for different API level intervals based on *π* (GenCCFGFromPIGraph in line 5) and performs static analysis to detect *use-without-def* (line 7) and *missing-key-APIs* (line 9) issues as discussed earlier (Section 5.2.1). If any callback compatibility issue is detected, Cider reports its issue location and issue type (lines 8 and 10).

Generating CCFGs from PI-Graph. To approximate the app control flows in different API levels and support callback compatibility issue detection, Cider generates app CCFGs based on PI-Graphs. Figure 6(a) shows the PI-Graph capturing the evolutions of the callback APIs that induced the WordPress issue in Figure 2. Given an app and a PI-Graph, Cider identifies all different API level intervals associated with the edges (e.g., [1, 22] and [23, 27] in Figure 6(a)). It also identifies the supported API levels of the given app by checking the attributes android:minSdkVersion and android:maxSdkVersion in the app's configuration file (i.e., AndroidManifest.xml). Cider then generates a CCFG for each supported API level interval *range* by traversing the PI-Graph along all edges whose associated API level interval is within *range* and adding the callback nodes associated with the edges. If a callback API is implemented in the app under analysis, the code in the method body will be added to the callback API's CCFG node. Figure 6(b) shows the CCFG for API level interval [23, 27], which is
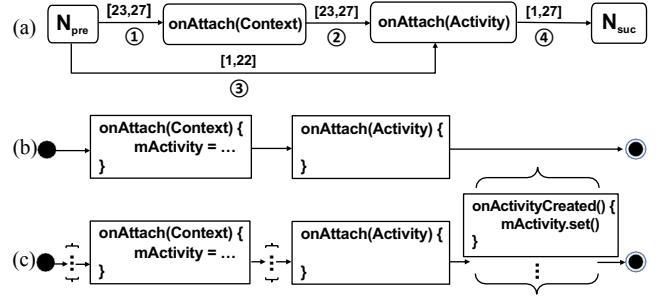
Figure 6: The steps of generating app CCFGs from PI-Graph. (a) is an example of PI-Graph related to the WordPress issue (Section 3). (b) is a simplified CCFG generated by Cider for WordPress. The CCFG describes the app's control flows among the callback APIs modeled by the PI-Graph in (a) at an API level in [23, 27]. (c) is an CCFG of the app after inserting the node of a callback API that is not in the PI-Graph in (a). The place of insertion is determined by the invocation order of the callback APIs specified in the API Reference.

generated with the PI-Graph in Figure 6(a) and the code snippet in Figure 2. Two callback API nodes are included in the CCFG as they can be reached along the edges whose associated API level is within [23, 27]. Since the API onAttach(Context) is implemented, its corresponding CCFG node contains the code in the method body.

So far, the generated CCFG only includes the callback APIs that are in PI-Graph models. Note that the control flow inconsistencies caused by callback API evolutions may propagate to other callback APIs that are not modeled by PI-Graphs. To analyze the dependencies between the callback APIs in PI-Graph models and other implemented callback APIs in an app, Cider further recovers the execution orders between the callback APIs in the generated CCFGs and other callback APIs. Specifically, we maintain a set of callback nodes for each edge in the CCFG. A node will be added to this set if its represented callback API can be executed between the callback APIs represented by the start and end nodes of the CCFG edge. For simplicity, we assume that the start and end nodes are both callback nodes. The cases where the start or end node is a helper node can be handled in a similar way. For the purpose of adding such callback nodes, we studied the specifications of all callback APIs defined in the classes that we extracted PI-Graphs from. For example, we extracted a rule from the API Reference that onActivityCreated should be executed after onAttach(Activity). Cider will then create a callback node for onActivityCreated and add it into the callback node set of the edge connecting onAttach(Activity) and N_{suc} as shown in Figure 6(c). One should note that the control flows between the callback APIs that are not in the PI-Graph models would stay unchanged at different API levels and thus may not induce callback compatibility issues. For this reason, we do not maintain orders between the set of callback nodes associated with CCFG edges. For those callback APIs of which the execution order cannot be inferred from the API specifications, Cider inserts them into all possible callback node sets in CCFGs.

Identifying callback compatibility issues. After generating CCFGs for different API level intervals, CIDER conducts control flow analysis to detect callback compatibility issues. It detects two common types of issues by analyzing each generated CCFG:

(1) *Use-without-def.* CIDER checks each CCFG and detects any variables that are used before being defined. Specifically, CIDER identifies the set of def and use statements for all callback nodes in each CCFG including those nodes in the callback node set of each edge. For each statement that uses a variable, CIDER checks whether the variable has been defined in any callback nodes that are (1) in the corresponding PI-GRAPH model and (2) executed before the use statement. If a variable is used without being defined, CIDER reports a warning of *use-without-def*.

(2) *Missing-key-APIs.* CIDER compares the invocations of key APIs in the list of CCFGs generated for different API levels. As discussed in Section 5.2 (the Bitmask issue), invoking such APIs will cause perceivable app behaviors such as popping up a notification. As a result, missing invocations of such APIs in certain API levels can induce inconsistent app behaviors that can be observed by users. CIDER will report a warning of *missing-key-APIs* if the invocations of key APIs, which are manually identified from our empirical study dataset, are inconsistent among CCFGs for different API levels.

According to our empirical study, the above two types of callback compatibility issues commonly occur in real-world Android apps. Therefore, we implemented two checkers for detecting these issues in the current version of CIDER. Our evaluation results show that the two checkers can already help detect a large number of real callback compatibility issues. In future, CIDER can be further extended with other checkers to detect more types of callback compatibility issues.

7 EVALUATION

We implemented CIDER on top of Soot [44], a static analysis framework for Android apps and Java programs. Our current implementation of CIDER encodes seven different PI-GRAPHs concerning 24 key APIs [10], which are extracted from our empirical study dataset. In this section, we evaluate CIDER with open-source Android apps. We aim to answer the following two research questions:

- **RQ3 (Effectiveness):** *Can CIDER effectively and precisely detect callback compatibility issues in real-world Android apps?*
- **RQ4 (Usefulness):** *Can CIDER provide useful information for Android app developers to facilitate the diagnosis and fixing callback compatibility issues?*

7.1 Experimental Setup

To answer the above research questions, we collected 20 open-source Android apps from GitHub that satisfy the following three constraints: (1) contain at least one commit after October 2017 (i.e., actively-maintained), (2) do not overlap with any projects selected for our empirical study, and (3) use at least one of the callback APIs in the seven PI-GRAPHs encoded in CIDER. We used the latest version of these 20 Android apps as our evaluation subjects to examine if CIDER can detect new callback compatibility issues in them. Table 2 gives the basic information of the 20 subjects, including: (1) the project name, (2) the app's category information provided by Google Play store (if applicable), (3) the revision used in our experiment, (4) the number of stars on GitHub, (5) the number of downloads

on Google Play store (if applicable), and (6) the number of lines of code. As shown in the table, the subjects are diversified, covering 10 different app categories. They are also non-trivial, containing thousands to hundreds of thousands of lines of code.

In the experiments, we configured CIDER to report the location and type (*use-without-def* or *missing-key-APIs*) of each detected issue. To evaluate the effectiveness of CIDER, we compared it with a baseline tool, Lint[4]. Lint is a static analyzer that scans the source files of Android apps to find common bugs, including callback compatibility issues. Since Lint is integrated in Android Studio, it has been widely-used by Android app developers to improve the software quality of their products. For comparison, we ran Lint on the same experimental subjects to detect potential callback compatibility issues. We manually checked the issues reported by CIDER and Lint to categorize them into true positives (TP) and false positives (FP). For fairness, we excluded those issues that are reported by Lint but are not related to any callback APIs in our extracted PI-GRAPHs. When categorizing each issue, we carefully examined the relevant source code. We would only categorize an issue as a true positive if it can cause the corresponding app to behave inconsistently at different API levels. To further evaluate the usefulness of CIDER, we reported the true issues detected by CIDER to the original app developers for confirmation and feedback. All experiments were conducted on an iMac with an Intel Core i5 CPU @3.4GHz and 16GB RAM.

7.2 Results of RQ3: Effectiveness of CIDER

As shown in Table 2, CIDER detected 14 issues in nine subjects, among which 13 are true positives (i.e., the precision is 92.9%). This shows that CIDER can precisely detect callback compatibility issues in Android apps. CIDER reported one false positive in Kolab Notes. We investigated this false positive to understand the limitation of CIDER. As discussed in Section 6, the CCFGs generated by CIDER do not capture the execution orders between callback APIs that are not in PI-GRAPHs (such orders would stay unchanged at different API levels). This can cause imprecision in control flow analysis. For instance, when analyzing Kolab Notes [17], CIDER reported a use-without-def issue because a variable defined in an evolved callback API was later used in a callback that handles a click event. The evolved API cannot be executed at some API levels and thus CIDER determines that the variable can be used without a definition and reported the issue. However, after manually reviewing this issue, we found that the app developers have already fixed this problem by including a definition of the variable in another callback API (not in the PI-GRAPHs), which can be called after the execution of the evolved API and before the execution of the click event handling callback. In our 20 evaluation subjects, CIDER reported only this false positive, suggesting that such imprecision in control flow analysis may not affect CIDER's effectiveness in practice.

Table 2 also presents the results of Lint. Lint detected two real callback compatibility issues but reported 19 false positives. The two true issues reported by Lint were also reported by CIDER. We investigated the 19 false positives and found that Lint simply reports the usage of deprecated callback APIs without analyzing whether such usage would really affect an app's runtime behavior. For example, in the app Calendula [8], Lint generated a warning suggesting

Table 2: Evaluation subjects and results.

No.	Project Name	Category	Last Commit	Stars	Downloads	KLOC	CIDER TP	CIDER FP	Lint TP	Lint FP	Issue	Status
1	AFWall+ [1]	Tools	71e6c66	1,100	500K+	21.8	1	0	0	0	786	fixed
2	Calendula [8]	Health	c476575	76	1K+	26.3	0	0	0	2	-	-
3	cccTV [9]	Education	667a83c	20	100+	7.8	2	0	2	0	8	confirmed
4	DuckDuckGo-Kotlin [11]	Tools	2d7d379	425	1M+	10.4	1	0	0	0	79	fixed
5	FOSS Browser [13]	-	e08f5b6	101	-	18.0	0	0	0	3	-	-
6	Kolab Notes [17]	Productivity	14ba3c3	42	1K+	73.4	0	1	0	0	-	-
7	MaterialFBook [18]	-	2cb3c61	90	-	68.0	0	0	0	0	-	-
8	Network-monitor [19]	Tools	0e17b95	54	50K+	20.8	0	0	0	0	-	-
9	NyaaPantsu [20]	-	53ad9a8	22	-	14.1	0	0	0	0	-	-
10	OONI Probe [22]	Tools	60cbd70	41	100K+	4.9	1	0	0	2	146	confirmed
11	OpenKeyChain [23]	Communication	7135525	1,001	100K+	848.5	0	0	0	1	-	-
12	OsmAnd [24]	Maps	b7a539f	1,410	5M+	662.5	1	0	0	5	4868	fixed
13	Padland [25]	Productivity	38f7e66	22	100+	58.9	1	0	0	2	47	fixed
14	PassAndroid [26]	Travel	2d50ff4	362	1M+	85.0	0	0	0	0	-	-
15	Ring [27]	Communication	20d4221	82	1M+	243.5	1	0	0	1	1831	not fixed
16	sg for SteamGifts [28]	Entertainment	59fe959	40	1K+	21.5	0	0	0	0	-	-
17	Simple-Solitaire [29]	Card	1483ee1	49	10K+	294.4	1	0	0	0	108	fixed
18	SuntimesWidget [30]	-	c9a3c00	24	-	63.1	0	0	0	2	-	-
19	SurvivalManual [31]	Books	13b1f43	326	1M+	49.4	0	0	0	0	-	-
20	Uber-ride [32]	-	4d77c38	209	-	12.7	4	0	0	1	105	fixed
	Total	-	-	-	-	-	13	1	2	19	-	-

that the use of callback API onReceivedError is not recommended as the API has been deprecated since API level 23. However, the use of this API does not affect the app's control flow at API levels higher than 23, while it is still necessary to use the API to support the app to run on lower API levels. In contrast, CIDER analyzes the control flow inconsistencies induced by the callback API evolutions. It is capable of precisely identifying the impact of an evolve callback API on an app's control flow.

> **Answer to RQ3:** CIDER can precisely detect callback compatibility issues in Android apps. With control flow analysis based on PI-GRAPH, CIDER can outperform Lint and effectively reduce the number of false positives.

7.3 Results of RQ4: Usefulness of CIDER

To evaluate the usefulness of CIDER, we reported the 13 true issues it detected to the app developers for their feedback. For each issue, we reported: (1) the issue location, (2) the root cause, and (3) potential patches for fixing the issue. To avoid overwhelming the app developers, for each concerned Android app, we submitted only one bug report including all true issues CIDER detected in the app. So far, we have received developers' replies for nine bug reports.

Eight of the nine (88.9%) bug reports (concerning 12 detected issues) have been confirmed by the app developers, among which nine issues mentioned in six bug reports have already been fixed. For example, for the app OsmAnd [24], we reported an issue (#4868) that a toast notification for displaying network errors will not be shown on devices with an API level lower than 23 because the functionality is implemented in a callback API that was not introduced until the API level 23. With the detailed information provided by us, OsmAnd

developers quickly realized the root cause of the issue and fixed the issue by following our suggestions. This shows that the callback compatibility issues detected by CIDER are useful for improving the software quality of Android apps. It also indicates that the knowledge learned from the 50 apps in our empirical study can be generalized to other Android apps.

The only bug report that was not confirmed by the app developers is Ring issue 1831 [27]. In this bug report, we reported an issue where a callback API cannot be invoked at API levels lower than 23. The callback API invokes another API that would change the title of an action bar widget. As a result, when the app runs on devices with an API level lower than 23, the title of the action bar widget will not be set properly. We considered this issue as a true positive. However, the app developers of Ring said that the issue we reported resides in some legacy code and the action bar widget is no longer displayed. In other words, even if the key API is not invoked in certain cases, it would not cause any perceivable consequences. This indicates that the determination of whether the missing invocation of a key API would really induce inconsistent app behaviors can depend on app-specific runtime contexts. It is a challenge for static analysis to derive such runtime contexts and determine perceivable impacts of invoking certain APIs. In future, we plan to explore the possibility of approximating such runtime contexts so that CIDER can better model app behavioral inconsistencies induced by the callback API evolutions. This will further improve the performance of CIDER.

> **Answer to RQ4:** Callback compatibility issues reported by CIDER are useful to Android developers. The knowledge learned from the 50 apps in our empirical study can be generalized to help improve the software quality of other Android apps.

8 DISCUSSIONS

Incompleteness of the key API list. CIDER takes a list of key APIs whose execution can cause user-perceivable outcomes as input to detect callback compatibility issues induced by the missing invocations of these APIs. In this paper, we manually built a list containing 24 APIs based on our empirical study dataset. The list is by no means complete. This may cause CIDER to miss real callback compatibility issues, leading to false negatives. In future, we plan to explore ways to automatically learn more key APIs from various sources such as API documentations to address this problem.

Timeliness of the empirical dataset. Since Android is evolving quickly, its framework code and official documents are updated constantly. Due to such evolutions, our empirical study dataset may become outdated someday. However, our main findings such as the control flow inconsistencies induced by the callback API evolutions may not easily get outdated.

Errors in manual inspection. Our study involved much manual work. For example, we manually identified callback APIs by analyzing API Reference. Such manual processes are subject to errors. This poses a threat to the validity of our findings. To mitigate the threat, two of the authors independently performed all manual inspections and cross-validated their results. We also release our dataset for public access [10].

9 RELATED WORK

9.1 Android Compatibility Issues

There are many existing studies on the evolution and fragmentation of the Android ecosystem as well as the resulting compatibility issues. Mutchler et al. [48] analyzed one million free Android apps and observed that compatibility issues induced by Android fragmentation are of serious concern to the entire app development community. McDonnell et al. [47] studied Android API evolution and its impact on API usage. Bavota et al. [36] explored how the fault-proneness and change-proneness of Android APIs can affect app ratings. Hora et al. [42] studied the developers' responses to the Android API evolutions. Fan et al. [39] investigated Android framework crashes and disclosed that one major cause for the crashes is API evolutions. Li et al. [45] studied deprecated Android APIs and the developers' reactions. Hu et al. [43] studied the compatibility issues in Android webview. However, none of the existing studies analyzed how Android evolutions can change callback APIs' invocation protocols and affect app control flows. In this paper, we systematically analyzed the changes in callback API invocation protocols and how such changes can cause compatibility issues.

Another work by Wei et al. [51] proposed a tool FicFinder to detect fragmentation-induced compatibility issues, which include compatibility issues induced by Android API evolutions (the focus of this paper). FicFinder can detect issues that are caused by invocations of certain issue-inducing APIs. It leverages backward slicing for issue detection but does not precisely analyze app control flows. In comparison, CIDER models inconsistent app control flows at different API levels and detects callback compatibility issues induced by such control flow inconsistencies. These issues cannot be detected by FicFinder.

9.2 Android Control Flow Analysis

Our technique analyzes the control flows of Android apps. There are several relevant techniques. For example, Blackshear et al. [37] proposed a technique that abstracts app control flows by discarding irrelevant flows while retaining the execution orders of event handlers that are useful for control flow analysis. Yang et al. [54] proposed the concept callback control flow graph (CCFG), which is also used in our work. They also proposed a technique to generate CCFGs for Android activity components by analyzing the invocation contexts of callback methods. Arzt et al. [34] designed FlowDroid, which performs control flow analysis of Android apps by synthesizing dummy main methods to model the executions of callback methods. Sun et al. [50] proposed a technique to detect code reuse by measuring the similarities of app control flow graphs. Wu et al. [53] designed a technique to detect unauthorized operations in Android apps by leveraging control flow analysis. Gordon et al. [40] proposed a static analysis tool DroidSafe that combines app runtime execution models and precise control flow analysis to detect the leakage of sensitive data in Android apps. Safi et al. [49] proposed Deva, a static analysis technique to detect anomalies induced by nondeterministic timing for triggering events. While all these techniques leveraged app control flow analysis, none of them considered the inconsistencies of app control flows at different API levels. To the best of our knowledge, our work is the first that focuses on studying how to model app control flow inconsistencies at different API levels and leverage the model to automatically detect callback compatibility issues.

10 CONCLUSION

In this paper, we conducted an empirical study to understand Android callback API evolutions. We found that the evolutions of Android callback APIs can cause significant changes in the APIs' invocation protocols. Such changes can affect app control flows and induce various compatibility issues. Based on the findings, we proposed a graph-based model, PI-GRAPH, to capture the changes of Android callback API invocation protocols. We further designed a static analysis technique CIDER that leverages the PI-GRAPH models to automatically detect two common types of callback compatibility issues. We implemented CIDER and evaluated its performance using 20 open-source Android apps. The results show that CIDER can precisely detect real and previously-unknown callback compatibility issues. In future, we plan to extend CIDER to support the detection of more types of callback compatibility issues. We also plan to enhance CIDER's performance by better approximating the impact of its detected control flow inconsistencies on app runtime behaviors.

REFERENCES

[1] 2018. AFWall. https://github.com/ukanth/afwall.
[2] 2018. Android API Differences Report. https://developer.android.com/sdk/api_diff/19/changes.html.
[3] 2018. Android Developer Dashboard. https://developer.android.com/about/dashboards/index.html.
[4] 2018. Android Lint. https://developer.android.com/studio/write/lint.
[5] 2018. Android Open Source Project (AOSP). https://github.com/aosp-mirror/platform_frameworks_base.
[6] 2018. API Reference | Android Developers. https://developer.android.com/reference/.
[7] 2018. BitMask. https://github.com/leapcode/bitmask_android.
[8] 2018. Calendula. https://github.com/citiususc/calendula.
[9] 2018. cccTV. https://github.com/stefanmedack/cccTV.
[10] 2018. Cider homepage. https://cideranalyzer.github.io.
[11] 2018. DuckDuckGo Android App. https://github.com/duckduckgo/Android.
[12] 2018. F-Droid. https://f-droid.org/en/packages.
[13] 2018. FOSS Browser. https://github.com/scoute-dich/browser.
[14] 2018. Fragment - Android Developer. https://developer.android.com/guide/components/fragments.html.
[15] 2018. GitHub. https://www.github.com.
[16] 2018. KeePassDroid Revision 467d6e8. https://github.com/bpellin/keepassdroid/commit/467d6e8.
[17] 2018. Kolab Notes. https://github.com/konradrenner/kolabnotes-android.
[18] 2018. MaterialFBook. https://github.com/ZeeRooo/MaterialFBook.
[19] 2018. Network Monitor. https://github.com/caarmen/network-monitor.
[20] 2018. NyaaPantsu. https://github.com/NyaaPantsu/NyaaPantsu-android-app.
[21] 2018. Omni-notes Issue 248. https://github.com/federicoiosue/Omni-Notes/issues/248.
[22] 2018. OONI-Probe for Android. https://github.com/TheTorProject/ooniprobe-android.
[23] 2018. OpenKeyChain. https://github.com/open-keychain/open-keychain.
[24] 2018. OsmAnd. https://github.com/osmandapp/Osmand.
[25] 2018. Padland. https://github.com/mikifus/padland.
[26] 2018. PassAndroid. https://github.com/ligi/PassAndroid.
[27] 2018. Ring. https://tuleap.ring.cx/projects/ring.
[28] 2018. Sg for SteamGifts. https://github.com/SteamGifts/SteamGifts.
[29] 2018. Simple Solitaire game collection. https://github.com/TobiasBielefeld/Simple-Solitaire.
[30] 2018. SuntimesWidget. https://github.com/forrestguice/SuntimesWidget.
[31] 2018. Survival Manual. https://github.com/ligi/SurvivalManual.
[32] 2018. Uber Rides Android SDK. https://github.com/uber/rides-android-sdk.
[33] 2018. WordPress Issue 6906. https://github.com/wordpress-mobile/WordPress-Android/issues/6906.
[34] 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. In PLDI. 259–269.
[35] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, and Michael D. Ernst Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents. In ASE. 669–679.
[36] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. TSE 41, 4 (Apr 2015), 384–407.
[37] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2015. Selective Control-Flow Abstraction via Jumping. In OOPSLA. 163–182.
[38] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In NDSS.
[39] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In ICSE. 408–419.
[40] 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.
[41] Hyung Kil Ham and Young Bom Park. 2011. Mobile Application Compatibility Test System Design for Android Fragmentation. In ASEA. 314–320.
[42] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Etien, and Stéphane Ducasse. 2018. How do Developers React to API Evolution? A Large-Scale Empirical Study. SOFTWARE QUAL J. 26, 1 (Mar 2018), 161–191.
[43] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. 2018. A Tale of Two Cities: How WebView Induces Bugs to Android Applications. In ASE.
[44] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot Framework for Java Program Analysis: a Retrospective. In CETUS.
[45] Li Li, Jun Gao, Tegawendé Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising Deprecated Android APIs. In MSR. 254–264.
[46] 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 FSE. 477–487.
[47] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In ICSM. 70–79.
[48] Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John Mitchell. 2016. Target Fragmentation in Android Apps. In SPW. 204–213.
[49] Gholamreza Safi, Arman Shahbazian, William GJ Halfond, and Nenad Medvidovic. 2015. Detecting Event Anomalies in Event-Based Systems. In FSE. 25–37.
[50] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. 2014. Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph. In IFIP. 142–155.
[51] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In ASE. 226–237.
[52] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission Evolution in the Android Ecosystem. In ACSAC. 31–40.
[53] Jianliang Wu, Tingting Cui, Tao Ban, Shanqing Guo, and Lizhen Cui. 2015. PaddyFrog: Systematically Detecting Confused Deputy Vulnerability in Android Applications. SECUR COMMUN. NETW. 8, 13 (Jan 2015), 2338–2349.
[54] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In ICSE. 89–99.