

A Tale of Two Cities: How WebView Induces Bugs to Android Applications

Jiajun Hu
The Hong Kong University of Science
and Technology
Hong Kong, China
jhuaoc@cse.ust.hk

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

Yepang Liu*
Southern University of Science and
Technology
Shenzhen, China
liuyup1@sustc.edu.cn

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

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

ABSTRACT

WebView is a widely used Android component that augments a native app with web browser capabilities. It eases the interactions between an app's native code and web code. However, the interaction mechanism of WebView induces new types of bugs in Android apps. Understanding the characteristics and manifestation of these WebView-induced bugs (ω Bugs for short) facilitates the correct usages of WebViews in Android apps. This motivates us to conduct the first empirical study on ω Bugs based on those found in popular open-source Android apps. Our study identified the major root causes and consequences of ω Bugs and made interesting observations that can be leveraged for detecting and diagnosing ω Bugs. Based on the empirical study, we further propose an automated testing technique ω DROID to effectively expose ω Bugs in Android apps. In our experiments, ω DROID successfully discovered 30 unique and previously-unknown ω Bugs when applied to 146 open-source Android apps. We reported the 30 ω Bugs to the corresponding app developers. Out of these 30 ω Bugs, 14 were confirmed and 7 of them were fixed. This shows that ω DROID can effectively detect ω Bugs that are of the developers' concern.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software testing and debugging**;

KEYWORDS

Empirical study, Android system WebView, GUI testing

ACM Reference Format:

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 *Proceedings of the 2018 33rd ACM/IEEE International Conference on*

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

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

© 2018 Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, September 3–7, 2018, Montpellier, France, <https://doi.org/10.1145/3238147.3238180>.

Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3238147.3238180>

1 INTRODUCTION

WebView [62] is a widely used Android component that allows Android apps to display webpages without relying on web browsers. It enables bi-directional interactions between the web end and the native end of an Android app. It is attractive to developers and companies since existing web technologies that work on other platforms can be readily integrated into a native Android app through WebView [53]. Earlier studies [81, 82, 87] have shown that WebViews are popularly used by apps in Google Play store [27].

Despite its usefulness, the interaction mechanism of WebView is complicated and causes various bugs in real-world Android apps. It is reported that apps containing WebViews are suffering from poor performance [33] and vulnerable to security attacks [28]. Existing studies focus mainly on the security issues induced by WebView [68, 77, 78, 81, 82, 87, 91, 93, 95, 97]. There is also an earlier study on detecting bugs in the WebView bridges between Java and JavaScript code in Android apps [77]. Nonetheless, these existing studies only explored a small fraction of WebView-induced bugs (ω Bugs for short) from limited perspectives. They did not aim to provide a systematic examination of such bugs. In fact, little is known about how WebView induces bugs to real-world Android apps.

To bridge the gap, we conducted the first systematic empirical study on 124 ω Bugs collected from popular open-source Android apps. In the study, we focused on the bugs that are related to the usages and runtime mechanisms of WebView. Well-studied bugs caused by common JavaScript errors [88, 89, 92] are excluded from our scope. Our study aims to understand ω Bugs from a wide range of perspectives by answering three research questions:

- **RQ1 (Bug cause):** *What are the common root causes of ω Bugs in Android apps? Can they be categorized?*
- **RQ2 (Bug consequence):** *What are the common consequences of ω Bugs in Android apps? How do ω Bugs affect user experience?*
- **RQ3 (Bug manifestation):** *How do ω Bugs manifest themselves? Can we propose testing techniques to effectively expose ω Bugs in Android apps?*

Answering these research questions is practically beneficial to both app developers and researchers. RQ1 characterizes the common root causes of ω Bugs and can guide developers to avoid such bugs at an early stage during an app’s development. RQ2–3 investigate the consequences and manifestation of ω Bugs and can guide test case generation to effectively detect ω Bugs.

In our study, we made several interesting observations. For example, we found that ω Bugs are mostly caused by (a) misalignments between a WebView’s lifecycle and an activity’s or a fragment’s lifecycle, and (b) the evolution of WebView. Only a few of our studied real-world ω Bugs are caused by the errors stemming from bridge communications [13]. We also found that the ω Bugs caused by WebView lifecycle misalignments often induce resource leakages and UI inconsistencies. This inspired us to design test oracles to facilitate the testing of ω Bugs.

Based on our empirical findings, we propose a technique, ω DROID, to effectively test common types of ω Bugs in real-world Android apps. ω DROID focuses on testing ω Bugs induced by WebView lifecycle misalignments by injecting lifecycle events when exercising WebView components. ω DROID is also equipped with effective test oracles derived from our empirical study to expose triggered ω Bugs. We applied ω DROID to 146 real-world Android apps. It successfully detected 30 unique and previously-unknown ω Bugs. We reported the 30 bugs to the corresponding app developers. So far, 14 of our reported bugs have been confirmed by the developers and seven of them have been fixed shortly afterwards. These results confirm the usefulness of our empirical study and show that ω DROID can effectively detect ω Bugs of developers’ concern. To summarize, this paper makes the following major contributions:

- We conducted the first comprehensive empirical study on ω Bugs in real-world Android apps. The study disclosed the characteristics of ω Bugs and identified potential research directions for the detection and diagnosis of ω Bugs. *We released our dataset to facilitate future research* [55].
- Driven by our empirical findings, we designed an automated testing technique ω DROID to effectively expose common types of ω Bugs in real-world Android apps. ω DROID injects lifecycle events into WebView test cases and exposes ω Bugs by leveraging effective test oracles.
- We evaluated ω DROID on 146 open-source Android apps. It successfully uncovered 30 ω Bugs. App developers showed interest in our detected bugs with 14 of them either confirmed or fixed.

2 BACKGROUND

2.1 Android System WebView

WebView [62] is an Android component that allows Android apps to display webpages. It differs from a web browser in that it provides a set of APIs to facilitate bi-directional interactions between the web end (JavaScript) and the native end (Java) of an Android app.

From Web to Native: A WebView can transfer control from the web end to the native end in two ways. The first way is to use *Bridge Communication* by creating a Java object that serves as a bridge to the JavaScript environment. This Java object is called *bridge object* (or *JavaScript Interface*). Developers can register a bridge object to the JavaScript environment using the `addJavaScriptInterface()`

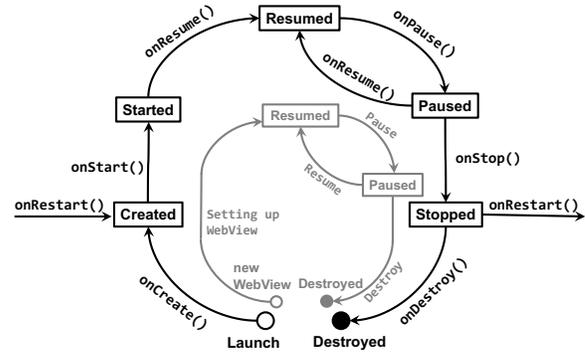


Figure 1: Activity and WebView lifecycles. Black color represents activity. Grey color represents WebView. Boxes represent states. Arrows represent state transitions.

API, which enables JavaScript code to invoke the public methods annotated by `@JavascriptInterface` in the bridge object. The second way is to register event handlers at a WebView to respond to events (e.g., URL loading) occurred at the web end. For example, one can override `shouldOverrideUrlLoading()` method to specify the task to be performed when a URL is being loaded by a WebView.

From Native to Web: The native end can transfer control to the web end using WebView APIs. For example, `evaluateJavascript()` asynchronously executes JavaScript code under the context of the current page displayed in a WebView.

2.2 Activity/Fragment and WebView Lifecycles

A WebView is often enclosed by an Android activity or fragment. An activity is a fundamental building block that provides Graphical User Interfaces (GUIs) to interact with an app, while a fragment represents a portion of a GUI and can be reused in multiple activities. Both activities and fragments are driven by a predefined lifecycle that specifies how they respond to events occurred at runtime. For example, the outer cycle of Figure 1 represents the lifecycle of an activity [4] whose states are traversed via invoking a set of callback methods. The runtime state of an activity is managed by the Android OS according to the occurring events.

A WebView’s lifecycle differs from that of its enclosing activity/fragment. It only has three states: *Resumed*, *Paused*, and *Destroyed*, as shown in the grey part of Figure 1. A WebView becomes focusable and displays its content when it is in the *Resumed* state. The resources loaded by a WebView such as video/audio, animation, geolocation are paused when WebView enters the *Paused* state. A WebView will be destroyed after entering the *Destroyed* state. Unlike an activity/fragment whose state transitions are managed by the Android OS, a WebView’s state transitions should be explicitly managed by app developers via invoking specific WebView APIs. For example, a WebView enters the *Paused* state after invoking `onPause()`, while it enters the *Destroyed* state after invoking `destroy()`. Developers may invoke multiple APIs in a state transition. For instance, developers who want to pause JavaScript execution before a WebView enters the *Paused* state can invoke two WebView APIs: `pauseTimers()` and `onPause()`.

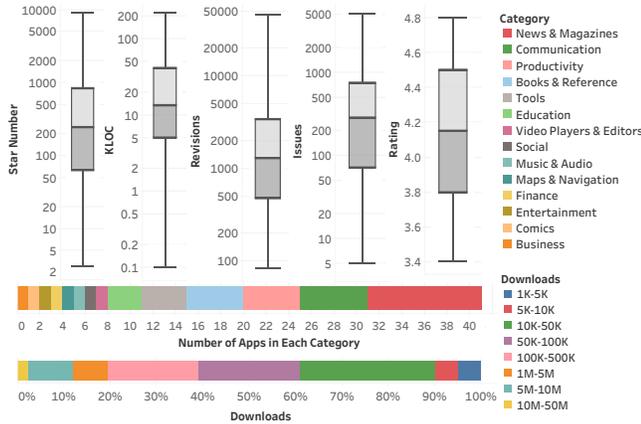


Figure 2: Statistics of apps used in our empirical study.

3 EMPIRICAL STUDY METHODOLOGY

We followed the process adopted by existing work [71, 79, 80, 98] to prepare the dataset for our empirical study. Specifically, we selected open-source Android apps as our study subjects because we need to dig into the apps’ documented bugs and corresponding code revisions to answer RQ1–3. We searched for suitable subjects by scanning all apps on F-Droid [20], a popular open-source Android app hosting site that is widely used by earlier studies [75, 96, 98]. We also included a set of open-source Android apps that were studied by previous work [67, 70, 72, 75, 90, 99] but not hosted on F-Droid. We considered an app to be a suitable subject if its source code contains at least one instance of the statement “import android.webkit.WebView;”, which indicates the use of WebViews. In total, we obtained 293 open-source Android apps for our study.

To locate ω Bugs in the 293 apps, we searched for code revisions and closed bug reports that contain keywords “webview” or “javascript” (case insensitive). 952 revisions and 1028 bug reports were returned by searching the keyword “webview”; 257 revisions and 347 bug reports were returned by searching the keyword “javascript”. The returned revisions and bug reports may overlap as a code revision and a bug report may contain both keywords. We took the following steps to filter out noises from the search results:

- We first excluded the code revisions and bug reports that are not related to valid ω Bugs. For example, some bug reports may be feature requests from users. Some code revisions and bug reports may accidentally contain our search keywords.
- For the bug reports that remained after the first step, we tried to recover the bug-fixing revision(s). A bug report was kept in our dataset only if we successfully found the bug-fixing revision(s).
- For the code revisions that remained after the first step, we only kept those that satisfy the following constraints: (1) we found the bug report(s) for the issues addressed in the code revisions, or (2) the commit logs or comments in code diff explicitly indicate that the code revisions are bug-fixing revisions.

Four of the co-authors were involved in the process for data collection, analysis, and cross-checking. As a result, we obtained 124 ω Bugs from 51 open-source Android apps. Figure 2 shows the statistics of these apps. Information including the number of stars, app

Table 1: ω Bugs root cause categories and their distributions.

Root Cause	Number
Misaligned WebView Lifecycles	35
WebView Evolution & Device Customization	40
Misconfiguration	21
API Misuse	16
Bridge Communication	4
WebView Limitation	5
Others	3
Total	124

size (in KLOC), number of revisions and issues are collected from the apps’ project hosting site such as GitHub [26]. Other information including the app rating, category, and number of downloads are collected from Google Play store [27] (10 of the 51 apps are not available on Google Play store). Such data show that our subjects are (1) large in size (around 10 KLOC on average), (2) well-maintained (containing thousands of revisions and hundreds of issues on average), (3) popular (90% of them received over 10K downloads), and (4) diversified (covering 14 categories). We carefully studied these ω Bugs by analyzing their bug reports, code revisions, and related discussions from online forums such as Stack Overflow [52] to answer RQ1–3. This took many rounds of investigations and discussions. In the end, the four co-authors reached consensus on the root cause, consequence, and manifestation for every ω Bug in the dataset.

4 EMPIRICAL STUDY RESULTS

4.1 RQ1: Bug Cause

We studied the patches and bug reports of the 124 ω Bugs in our dataset to understand their root causes. We identified six major categories of root causes from these ω Bugs. Table 1 lists the categories and the number of bugs in each category. In the following, we discuss the common root causes with examples.

4.1.1 Misaligned WebView Lifecycles. 35 of the 124 ω Bugs are caused by the misalignment between a WebView’s lifecycle state and that of its enclosing activity/fragment. We call this phenomenon *lifecycle misalignment*. When the enclosing component of a WebView enters a certain lifecycle state, it expects the enclosed WebView to enter a corresponding lifecycle state. However, since the lifecycle of a WebView and that of its enclosing component are separately maintained by app developers and Android OS, lifecycle misalignments can easily arise, inducing ω Bugs. For example, WordPress contained a bug (issue 484 [64]) that caused users to keep hearing video sounds after they left an activity enclosing a video-playing WebView. The bug occurred because developers did not properly make the WebView enter *Paused* state when its enclosing activity is paused. To fix this bug, developers aligned the lifecycle of the WebView with that of its enclosing activity, as shown in Figure 3, by explicitly calling `mWebView.onPause()` (line 5) in the `onPause()` callback of `WebViewActivity`.

4.1.2 WebView Evolution & Device Customization. ω Bugs can experience severe compatibility problems. These bugs occurred because

```

1. public class WebViewActivity extends WPActionBarActivity {
2. + @Override
3. + protected void onPause() {
4. +     super.onPause();
5. +     mView.onPause();
6. + }
7. }

```

Figure 3: Patch for WordPress issue 484 (Simplified)

WebView is fast evolving and non-uniformly supported on different device models, causing a WebView to behave inconsistently across WebView versions and devices. We identified 40 such ω Bugs.

In Android ecosystem, the WebView package is maintained and upgraded separately from Android OS since Android 5 [6]. This aggravates the situation of compatibility problems [74, 98] because ω Bugs can occur not only when the Android OS upgrades but also when WebView itself is upgraded. We observed 33 such bugs in our dataset. For example, Gadgetbridge issue 758 [24] was caused by an upgrade of WebView to version 60 that stopped supporting top-frame navigations to data URLs because such navigations were widely used in spoofing and phishing attacks [43]. App developers have to modify their code to adapt to the change.

Besides, WebView is not supported uniformly across device models. As a result, ω Bugs can be device-specific. We observed 7 such bugs in our dataset. For example, Aard Dictionary suffered from a bug (issue 28 [3]) that caused the page scrolling in a WebView on e-ink devices to create a lot of screen refreshing and annoying flickers. The bug occurred because the high display latency of e-ink devices makes it hard to guarantee the smoothness of page scrolling [1].

4.1.3 Misconfiguration. Many webpages require customized WebView settings to function properly. For example, displaying a webpage with JavaScript content requires JavaScript to be enabled in WebView settings. Misconfigured WebView settings can prevent a webpage from being loaded and displayed correctly. However, properly configuring WebViews is non-trivial because there are over 50 configurable parameters in WebSettings [61], and in many cases the webpages to be loaded at runtime may not be known in advance when an app is developed. 21 of the 124 ω Bugs occurred due to the misconfiguration of WebView settings. For instance, four bug reports in Nextcloud News Reader [34] mentioned that articles are displayed using desktop configuration with large screen widths (Figure 6a). This would significantly reduce the readability as users have to frequently scroll their screens from left to right to read an article. The problem occurred because the WebView in the app failed to redirect the requests to mobile sites. Such redirection relies on the web storage capability that needs to be explicitly enabled in WebView settings. The developers of the app fixed the bug by properly configuring the WebView, i.e., enabling web storage by invoking `WebSettings.setDomStorageEnabled(true)`.

4.1.4 API Misuse. 16 of the 124 ω Bugs were caused by the misuses of WebView APIs. While WebView provides powerful APIs for bi-directional interactions between an app's web end and native end, the protocols of these APIs are complex. Without good understanding of the constraints imposed by these protocols, developers can easily make mistakes when using WebView APIs. An example

```

1. + final WebView webView =
2. +     (WebView) getView().findViewById(R.id.viewPostWebView);
3. + webView.setWebViewClient(
4. +     new WPWebViewClient(WordPress.getCurrentBlog()));
5. + new Thread() {
6. +     @Override
7. +     public void run() {
8. -         final WebView webView =
9. -         (WebView) getView().findViewById(R.id.viewPostWebView);
10. -         webView.setWebViewClient(
11. -         new WPWebViewClient(WordPress.getCurrentBlog()));
12.             //...
13.     }
14. }.start();

```

Figure 4: Patch for WordPress issue 1057

of such constraints is that WebView APIs must be called on an app's UI thread. According to WebView documentation [31]: "if you call methods on WebView from any thread other than your app's UI thread, it can cause unexpected results". However, we observed 7 cases where this protocol is violated. For instance, Figure 4 shows the patch for WordPress issue 1057 [64]. In the buggy version, WebView APIs were mistakenly called on a non-UI thread (lines 8–11). Developers fixed the bug by relocating the API calls to the UI thread (lines 1–4). Another example of the constraints is the *timeliness of WebView API invocation*. Invocation of some WebView APIs is subject to time constraints. Violations of these constraints can result in runtime errors. For instance, the API `getUrl()` will return null if webpage loading has not yet finished. This can easily lead to `NullPointerException`. Slide suffered from app crashes (issue 2540 [49]) due to the premature invocation of `getUrl()`.

4.1.5 Bridge Communication. Previous work [77] has shown that the bridge communication between Java and JavaScript is error-prone. We observed four bugs related to bridge communications in our dataset. For example, two performance bugs in Wikipedia [63] occurred due to heavy transactions across the bridge. This forced the system to allocate a huge amount of memory for data communications across the bridge, thereby increasing the chances of stack overflow and out of memory errors. The other two bugs occurred because of obfuscation. When the bridge object methods are obfuscated, they cannot be called by name from JavaScript.

4.1.6 WebView Limitation. Although WebView is a powerful component, it has limitations. Due to security concerns, WebView cannot support all web browsing features. As a result, developers may encounter problems when they implement features that WebView cannot support. We observed five such ω Bugs. For example, several bug reports in WordPress [64] and RedReader [42] mentioned that WebViews showed a blank page when opening file download links (e.g., pdf), which is not supported by WebView. Developers fixed these bugs by launching web browsers to handle the links.

Finally, there are other types of bugs in our dataset. For example, OCREader [35] issue 9 reported a bug that a WebView displays blank pages when it is wrapped by a `NestedScrollView` because it has to load the whole page at once, thus exceeding graphics memory limit. However, such bugs are rare and we do not further discuss them in this paper.

```

1. public class AboutActivity extends ActionBarActivity {
2. + @Override
3. + protected void onDestroy() {
4. +     super.onDestroy();
5. +     if(webViewContainer != null && webView != null) {
6. +         webViewContainer.removeAllViews();
7. +         webView.destroy();
8. +     }
9. + }
10. }
    
```

Figure 5: AntennaPod revision 77647cc (Simplified)

Answer to RQ1: We identified six categories of common root causes for ω Bugs in Android apps, where misaligned WebView lifecycles and WebView evolution & device customization are two dominant ones. Only a few of our studied ω Bugs are caused by bi-directional interactions across WebView bridge objects.

4.2 RQ2: Bug Consequence

To understand the common consequences of ω Bugs, we analyzed the bug reports, follow-up discussions, commit logs, and correlated code comments of the ω Bugs in our dataset. For ω Bugs that we cannot identify consequences from their textual descriptions, we manually reproduced them in emulators to observe its consequences at runtime. In total, we successfully identified the consequences of 114 ω Bugs. We failed to reproduce the remaining 10 bugs due to lack of configuration details or necessary environments. We observed the following common types of consequences caused by ω Bugs.

4.2.1 Performance Degradation. 31 of the 124 ω Bugs induced performance degradations. These bugs demonstrate similar consequences of conventional performance bugs [79]. They can cause resource leakages, battery drains, app slowdowns, etc. For example, AntennaPod revision 77647cc [9] (Figure 5) fixed a WebView-induced memory leakage. The bug occurred due to the lifecycle misalignment problem (Section 4.1.1): when destroying the enclosing activity of a WebView, developers forgot to destroy the enclosed WebView. This would prevent the runtime in recycling the resources used by the WebView and its enclosing activity, causing memory leakage. The bug was fixed by adding an explicit call to the WebView’s destroy() API (line 7) in the onDestroy() callback of the enclosing AboutActivity.

4.2.2 Problematic UI Display. 29 of the 124 ω Bugs resulted in problematic UI displays in WebViews. Since WebViews are intensively used to display webpages in Android apps, such problems can significantly affect app usability such as the example (Figure 6a) discussed in Section 4.1.3. WebView may also display web content incorrectly. For example, Figure 6b shows a WebView displaying garbled characters (PocketHub issue 427 [38]), which was induced by WebView API evolution.

4.2.3 UI Inconsistency. Another 17 of the 124 ω Bugs exhibit UI inconsistencies after a restart of the WebView’s enclosing activity/fragment. For example, in Wikipedia bug report T63512 [63], users complained about the inconsistent display of webpages: If the enclosing activity of a WebView is restarted after users scroll down a webpage, the webpage will be reloaded and the users’ scroll



(a) Nextcloud News Reader (b) PocketHub

Figure 6: Problematic UI displays of WebView

position will get lost. Such problems are common since events that induce restart of activity/fragment (e.g., device rotation, exiting and re-entering an app, etc) are commonly triggered by app users.

For the remaining 37 bugs, 15 of them induce crashes (e.g., Slide issue 2540 [49]) and the rest 22 bugs are app-specific functional issues. We can observe from the above discussions that the majority of our studied ω Bugs cause performance or UI issues. Generally, it is non-trivial to define test oracles to detect performance and UI bugs via testing [79, 86]. However, studying the consequences of ω Bugs enables us to propose mechanisms to generate test oracles for common ω Bugs, which will be elaborated in the next section.

Answer to RQ2: Major consequences caused by ω Bugs include performance degradation, UI issues, and app crashes. These consequences can lead to poor user experience.

4.3 RQ3: Bug Manifestation

Understanding the manifestation of ω Bugs can help us design and select effective tests to detect ω Bugs. In RQ3, we aim to investigate common inputs required to trigger ω Bugs so as to design effective WebView testing techniques. We studied the bug reports, code revisions and related discussions of the ω Bugs in our dataset to learn how these bugs were triggered and noted by app developers. Similar to what we did when studying ω Bugs consequences, we manually reproduced the ω Bugs, for which we fail to learn the manifestation from textual data sources. In total, we successfully understood the manifestation of 97 of the 127 ω Bugs. The remaining bugs cannot be reproduced due to reasons similar to those discussed in Section 4.2. In the study, we made four major observations, which demonstrate common challenges in testing ω Bugs.

4.3.1 Manifestation of ω Bugs can be content-sensitive. The manifestation of 30 ω Bugs requires webpages with specific content such as images (e.g., AnkiDroid issue 2824 [8]), animations (e.g., Nextcloud News Reader issue 331 [34]), video/audio (e.g., WordPress issue 484 [64], see Section 4.1), and various character encodings (e.g., PocketHub issue 427 [38]). For example, WordPress issue 484 [64] (Figure 3) can only be exposed by webpages with video/audio content. However, webpage contents can be highly diversified while an ω Bug can only be triggered by specific contents. Therefore, how

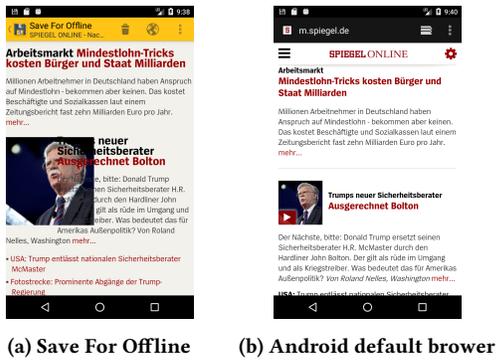


Figure 7: UI display differences between WebViews and mobile web browsers

to select webpages with the specific content to effectively expose ω Bugs is a critical challenge for WebView testing.

4.3.2 Lifecycle events are required to trigger common ω Bugs. In our dataset, 27 ω Bugs require specific sequence of lifecycle events to trigger. For example, WordPress issue 484 (Figure 3) can only be triggered when the enclosing activity of the WebView is created and paused. The bug in AntennaPod (Figure 5) can only be triggered when the enclosing activity of the WebView enters the *Destroyed* state. This observation indicates that generating sequence of lifecycle events is often needed when testing ω Bugs.

4.3.3 Effective oracles can be proposed for automated detection of ω Bugs. The ω Bugs in our dataset were confirmed mainly based on developers' manual judgement. We carefully studied the measures developers took to determine the existence of these ω Bugs and examined if the measures can be generalized as test oracles for ω Bugs. As a result, we made two observations.

First, 25 of the 124 ω Bugs can be identified by comparing app UI displays or resource usages before and after triggering lifecycle events. As discussed in Section 4.1, ω Bugs are commonly caused by the misalignment between the lifecycle state of a WebView and that of its enclosing activity/fragment. We observed that the majority (23 / 35) of such bugs can be identified by comparing app UI displays or resource usages before and after triggering lifecycle events. For example, Wikipedia issue T63512 discussed in Section 4.2.3 can be detected via comparing the app UI displays before and after recreating the WebView's enclosing activity (no matter which part of a webpage was originally displayed by the WebView on the screen, the top of the webpage would be displayed after activity recreation). WordPress issue 484 discussed in Section 4.1 is another typical example that can be detected by comparing the video resource usages before and after pausing the WebView's enclosing activity (the video would continue to play even when the activity is paused). This observation motivates us to leverage such comparisons as test oracles to detect ω Bugs that are caused by misaligned WebView lifecycles (Section 5).

Second, another 12 of the 124 ω Bugs can be identified by comparing the behaviors of WebViews and web browsers. Some app developers took web browsers' behaviors (e.g., Chrome and Firefox) as an oracle to identify abnormal webpage display issues. For

example, an ω Bug in Save for Offline [45] that displays overlapped content can be identified by comparing the page displayed by the WebView and that displayed by the Android default browser, as shown in Figure 7. This suggests the possibility of leveraging differential testing to detect ω Bugs.

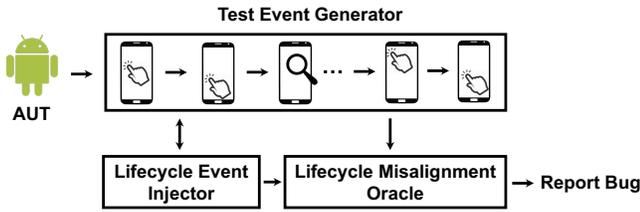
4.3.4 Conventional test coverage criteria may not apply to ω Bug testing. Unlike a native app, the program logic of an app using WebViews spans across its web code and native code. For example, a popular game 2048 [2] only contains one activity and 136 lines of Java code. Its complex game functionalities are implemented in JavaScript. As a result, conventional test coverage criteria (e.g., code coverage or activity coverage) leveraged by existing test generation techniques [66, 84, 85, 96] may not be effective for testing ω Bugs. In particular, JavaScript code coverage may not be applicable because (1) the manifestation of many ω Bugs does not require the execution of JavaScript code and (2) it is hard to measure the total number of JavaScript code since a WebView can display an unlimited number of webpages. To effectively test ω Bugs, new criteria measuring the coverage of different WebView states and WebView's interactions with app native components are desirable. Future research can focus on designing such coverage criteria to aid ω Bugs testing.

Answer to RQ3: ω Bugs's manifestation highly depends on the content of the loaded webpages and often requires the intervention of lifecycle events. Automated oracles and new coverage criteria are needed to effectively test apps using WebViews.

5 TESTING ω BUGS

Driven by our empirical findings, we propose an automated testing technique ω DROID to detect common types of ω Bugs in real-world Android apps. The technique is designed based on two major observations (section 4.3): (1) lifecycle events are required to trigger common ω Bugs, and (2) ω Bugs can be identified by comparing UI displays and resource usages before and after triggering lifecycle events. Inspired by the first observation, ω DROID generates test cases by injecting predefined lifecycle event sequences on-the-fly when the execution of an app reaches a screen containing WebView components. ω DROID then leverages the second observation to compare app UI displays and resource usages before and after the app handles the injected events to identify ω Bugs.

Figure 8 presents a high-level overview of ω DROID, which consists of three modules: *Test Event Generator*, *Lifecycle Event Injector*, and *Lifecycle Misalignment Oracle*. *Test Event Generator* explores the functionality of an app under test (AUT) by dynamically generating user and system events. During exploration, it detects the existence of WebViews in the displayed components. If a WebView is detected, it analyzes the UI hierarchy and builds a UI model for the loaded webpage and invokes *Lifecycle Event Injector*. *Lifecycle Event Injector* then triggers predefined lifecycle events sequence and invokes *Lifecycle Misalignment Oracle* to detect ω Bugs. We now discuss the details of each module.

Figure 8: Overview of ω DROID

5.1 Test Event Generator

The *Test Event Generator* module generates test events for a given AUT. It exercises the AUT using existing test generation techniques for Android apps and builds UI models of WebViews for further analysis. We implemented ω DROID on top of Monkey [7] while it can be adapted to work with other test generation approaches [66, 69, 73, 76, 83–85, 96]. To detect the existence of WebViews, *Test Event Generator* analyzes the UI hierarchy of the screen right after the AUT handles each triggered event. If WebViews exist in the current screen, it builds a UI model for the loaded page. The UI model will be used later by the *Lifecycle Misalignment Oracle* module to detect ω Bugs. The UI model of a webpage is a DOM tree (illustrated in Figure 9) extracted using UiAutomation [57]. In the UI model, leaf nodes represent elements in the webpage that are displayed to user and non-leaf nodes indicate how leaf nodes are organized in the WebView. Each node has a set of attributes: *class name, content description, checkable, checked, clickable, enabled, focusable, focused, scrollable, long-clickable, password, and selected*.

Challenge: A key challenge in *Test Event Generator* is to determine the time to build the UI model when a webpage is loaded. The model should be built after the page is fully rendered; otherwise the built model would be incomplete and cause imprecision in later steps. However, the loading time of a page is non-deterministic, depending on the page content and network speed.

Solution: We addressed the challenge by leveraging two common scenarios in which a page’s DOM structure changes when it is being loaded. In the first scenario, a WebView displays either a blank page or a simple “loading” message until it has fully loaded the page. In this case, we have a trivial DOM structure for a while in the beginning, followed by a steady non-trivial DOM structure after the page has been loaded. In the second scenario, a WebView renders a page progressively with light content (e.g., text) displayed before heavy content (e.g. images and videos). In this case, we have an evolving DOM structure that gradually becomes non-trivial and steady after the page has been loaded. With these observations, ω DROID delays the building of a UI model for a page until it detects the existence of a steady non-trivial DOM structure or timeout. If such DOM structure is detected, the current DOM tree will be treated as the UI model for the loaded page. Note that ω DROID monitors the changes of DOM structure instead of the changes of DOM contents as the loaded contents may be dynamic (e.g., animations).

5.2 Lifecycle Event Injector

The *Lifecycle Event Injector* module injects three kinds of lifecycle event sequences (they often trigger ω Bugs according to our empirical study) when the execution of the AUT reaches a screen

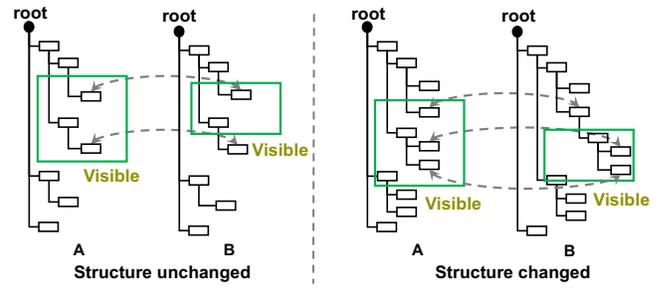


Figure 9: The UI model of a webpage loaded by WebView and the oracle to detect UI display inconsistency. A and B represent the UI models of the webpage before and after activity restart. The green box represents the part of webpage that is visible to users. The dashed lines represent that a node in A is also found in B.

containing WebViews. These event sequences can cause an activity to go through the following state changes: (1) paused and then resumed, (2) stopped and then restarted, and (3) destroyed and then recreated. By injecting such event sequences, ω DROID can explore all the lifecycle states of a WebView’s enclosing activity/fragment and simulate common user interactions on the app such as turning screen off and on, pressing home button and re-entering the AUT, and rotating devices. After the AUT handles each injected lifecycle event sequence, ω DROID rebuilds a UI model for the webpage being displayed in the WebView using the mechanism discussed in Section 5.1. The model is then fed into the *Lifecycle Misalignment Oracle* module to detect ω Bugs.

5.3 Lifecycle Misalignment Oracle

The *Lifecycle Misalignment Oracle* module relies on two rules to detect ω Bugs. The rule for detecting UI inconsistencies is based on a common sense expectation [100] that the displayed content of a WebView after activity restart should be consistent with that before the restart. The rule for detecting unpaused resources is based on the expectation that when the enclosing activity is paused, resources loaded by WebViews should be paused as well. The current version of ω DROID only detects unpaused video resources. We leave the detection of other kinds of resources as future work. ω DROID reports an ω Bug if any of the two rules is violated.

Challenge: While the detection of unpaused video resources can be implemented by testing sound from video after the WebView’s enclosing activity is paused, it is non-trivial to define an oracle to detect UI inconsistencies of WebViews. Checking the differences of screenshots [100] is inadequate. As explained in Section 5.1, many webpages contain elements with dynamic effects such as animations or videos. Such elements can result in different screenshots even when the webpage contents are the same. Furthermore, some apps may handle activity restarts by saving and restoring the scroll positions of the displayed webpages. The restored scroll positions can be slightly different from the original ones before the activity restarts. Such differences are usually tolerable by users but screenshots are sensitive to subtle changes. As a result, simply comparing screenshots may lead to false positives in ω Bugs detection.

Solution: To precisely detect ω Bugs, *Lifecycle Misalignment Oracle* module compares the webpages' tree-based UI models before and after the AUT handles the injected lifecycle events. Such UI models ignore the dynamic effects of webpage elements and thus help reduce false positives in bug detection. The comparison oracle is designed based on the intuition that *all the previously-displayed content in a WebView should be found in the page after the WebView's enclosing activity restarts, and at least part of the previously-displayed content should still be displayed after the activity restarts.*

Mapping UI nodes. Figure 9 illustrates how to compare the tree-based UI models. The models with label A and B represent the UI models of the webpages before and after the enclosing activity restarts. To compare the UI models, *Lifecycle Misalignment Oracle* first maps the visible nodes in A with nodes in B. This is necessary because the elements in the displayed webpages do not have unique identifiers. The mapping is based on the tree structure of the UI models and the attributes contained in each node. If the structure of a loaded webpage does not change, a node n in A is first compared with a node n' in B that is located at the corresponding position in the UI model. Node n is successfully mapped to node n' only if n' shares the same attribute values with n . If the structure of the webpage changes, we cannot pair nodes according to their positions because one node in A may be placed at another position in B. In such cases, *Lifecycle Misalignment Oracle* pairs the nodes in A and B according to their attributes. A node in A is successfully found in B if there exists a node in B whose attribute values are the same with that of the node in A.

Reporting inconsistencies. After mapping the nodes between UI models, *Lifecycle Misalignment Oracle* will report UI inconsistencies if any visible node in A cannot be mapped to nodes in B or all visible nodes in A become invisible in B.

6 EVALUATION

In this section, we evaluate the ability of ω DROID in detecting real-world ω Bugs by answering two research questions:

- **RQ4 (Effectiveness and Practicality of ω DROID):** *Can ω DROID effectively detect ω Bugs in real-world Android apps of concern to app developers?*
- **RQ5 (Usefulness of Lifecycle Events and Test Oracles):** *Are the lifecycle events injected by ω DROID and the test oracles adopted by ω DROID useful to identify anomalous app behaviors induced by ω Bugs?*

6.1 Experimental Subjects and Setup

We conducted evaluation based on the 293 apps selected in Section 3. Among them, we excluded the 51 apps which were used to form our empirical study dataset. We further filtered out those apps that do not have any code revisions within the past six months for evaluation. This is because we are only interested in actively-maintained apps since answering RQ4 requires developers' feedback. As a result, we obtained 146 apps for the experiments. We ran ω DROID on the latest version of these apps to check if ω DROID can detect new ω Bugs in them. We followed the practice adopted by an existing study [85] to set up our experiments and ran ω DROID for one hour to test each subject. The tests were executed on Android emulators

running Android 6.0 (Marshmallow), which was the most popular Android system version [5] at the time of our experiments.

To answer RQ5, we built two baseline methods for comparisons:

- **Baseline I: Test Event Generator:** This baseline leverages *Test Event Generator* to trigger events and detect ω Bugs based on software crashes (a common test oracle used in existing studies [85, 96]).
- **Baseline II: Test Event Generator + Lifecycle Event Injector:** Comparing to Baseline I, Baseline II additionally leverages *Lifecycle Event Injector* to inject lifecycle events when app subjects are being executed. It also detects ω Bugs based on software crashes.

For these two baselines, we reported the number of unique crashes following the approach adopted by Su et al. [96]. We did not prepare an additional baseline in which only *Lifecycle Misalignment Oracle* is kept because our proposed oracle targets UI inconsistencies occurred between lifecycle events, thus it cannot work without *Lifecycle Event Injector*.

We manually inspected all ω Bugs reported by ω DROID to categorize them as true positives or false positives. For each detected ω Bug, ω DROID outputs the screenshots, the name of foreground activity when the bug is detected, and an event trace to trigger the bug. With such information, we reproduced each ω Bug and categorized it as a true positive only if the oracle violation was induced by the inserted lifecycle events. As such, an oracle violation on a webpage containing dynamic elements will be treated a false positive because the page content can become "inconsistent" even without triggering lifecycle events. We further reported the ω Bugs that are categorized as true positives to the app developers for confirmation. The experiments were conducted on a PC with an Intel Core i5 CPU @3.2 GHz and 8GB RAM.

6.2 Results for RQ4

ω DROID reported ω Bugs in 31 of the 146 apps. Table 2 shows the results for these 31 apps, including app name, KLOC, number of TPs, number of FPs, and the IDs of our submitted bug reports. Note that multiple failures can be induced by the same ω Bug. We consider two failures as duplicates (caused by the same ω Bug) if they are manifested on the same WebView of the same activity/fragment with the same consequences. For example, ω DROID reported two failures in the same WebView that loads log-in pages of different social network websites when testing Twidere [56]. Although the loaded webpages are different when the two failures occurred, the failures were considered as duplicates since they demonstrate the same consequences that the webpage reloads without restoring its previous states after recreating the enclosing activity.

In total, ω DROID reported 36 unique bugs and 30 of them are TPs (83.3% precision). All the three types of event sequences that can be injected by *Lifecycle Event Injector* exposed ω Bugs that were categorized as TPs. For example, Forecastie [21] uses a WebView to load a weather map. ω DROID detected a bug (issue 275) that any navigations on the map will be lost after triggering lifecycle events that destroy and recreate the enclosing activity. Another example is Reddinator [41]: the video in a WebView does not stop even after the enclosing activity is sent to the background.

We inspected the six FPs reported by ω DROID and identified two major reasons for the imprecision. Four of the FPs were due

Table 2: Evaluation Result of ω DROID. Bug ID with superscript “f” means that the bug has been fixed, with superscript “c” means that the bug has been confirmed, and with superscript “n” means that developers decided not to fix the “bug”.

Id	App Name	KLOC	TP	FP	Bug ID(s)	Id	App Name	KLOC	TP	FP	Bug ID(s)
1	Serval Mesh [46]	112	1	0	#136	17	Cherry [14]	1.9	1	0	#7 ^c
2	arXiv mobile [10]	3.4	1	0	#16	18	Termux [54]	9.3	1	0	#664
3	EteSync [19]	12.5	1	0	#28 ^f	19	Kimai [29]	1.9	1	0	#30 ^c
4	dandelion* [16]	8	0	1		20	Barcode Scanner [11]	43.2	1	0	#989 ⁿ
5	Twidere [56]	19.4	1	1	#1110	21	ForRunners [22]	5.1	0	1	
6	Budget Watch [12]	7.6	1	0	#159 ^f	22	sg [47]	13.2	1	0	#64 ^c
7	Rental Calc [44]	6.7	1	0	#40 ^f	23	Shortyz [48]	16.9	3	0	#134,#135
8	Vanilla Music [58]	20.4	1	0	#767 ⁿ	24	Crossword [15]	1.1	1	0	#2 ^c
9	Forecastie [21]	3	2	0	#274,#275	25	Gift Card Guard [25]	1.6	1	0	#42 ^f
10	Snapcast [51]	17.6	1	0	#384	26	OpenVPN for Android [36]	60.6	1	0	#864 ^c
11	Vespucci [59]	71.2	1	0	#691 ^f	27	Web Opac [60]	36.9	0	1	
12	RasPi Check [40]	7.7	1	0	#179 ^c	28	MTG Familiar [32]	21.1	1	0	#373 ⁿ
13	Freifunk Auto Connect [23]	2.5	1	0	#20	29	LinCal [30]	2.5	1	0	#14 ^c
14	Padland [37]	4.6	0	1		30	Diary [17]	2.6	2	0	#47 ^f , #52 ⁿ
15	Polar Clock [39]	0.2	0	1		31	Drinks [18]	1.3	1	0	#72
16	SMS Backup+ [50]	11.2	1	0	#878 ^f	Total Num of TPs: 30; Total Num of FPs: 6					

to long page loading time, which caused timeout in the UI model generation process and thus ω DROID built incomplete UI models for the webpages. For example, the bug in Padland [37] was reported because ω DROID compared two incomplete UI models before and after injecting lifecycle events. The other two FPs were induced by dynamic content in webpages. For example, Polar Clock [39] uses a WebView to load a webpage containing a timer, which constantly changes over time. As a result, the UI model of the webpage is simultaneously changing and induces FPs.

To evaluate the practicality of ω DROID, we reported the 30 TPs found by ω DROID to the app developers for their feedback. In each bug report, we provided the reproduction steps, screenshots, and possible solutions. So far, we have received developers’ replies for 18 of the reported bugs. Developers confirmed 14 of the 18 bugs and quickly fixed seven of them. They also provided positive feedback to our reported bugs. For example, developers of Kimai [29] responded “*Good catch! Having always rotation lock on so didn’t quite get that yet. Happy about PRs (pull requests);*”. The comment shows the developers’ interest on the reported bug but they failed to identify it as they did not inject lifecycle events when testing the app. This shows that ω DROID can detect unknown ω Bugs that are of developers’ concern. For the other four bugs, three of them were considered not harmful by the developers and therefore there is no need to fix them. One of them (MTG Familiar [32]) was acknowledged by developers but they would not fix it because WebViews will be replaced in their app. Note that our oracle is based on common expectations of app responses to lifecycle events [100]. There may be exceptional cases that app developers consider violating such expectations is tolerable. However, this is not common in our evaluation as we only observed three cases.

Answer to RQ4: ω DROID can effectively detect ω Bugs in Android apps with high precision. ω DROID is practically useful and can find ω Bugs that are of developers’ concern.

6.3 Results for RQ5

To show the usefulness of the injected lifecycle events and our proposed oracles, we compared the results of ω DROID and the two baseline methods. Since crash is a general oracle that can help identify bugs beyond ω Bugs, when evaluating the baseline methods we only counted the crashes related to WebView. We consider a crash related to WebView if the crash notification message mentions WebView or there are WebView APIs in the crash stack.

We observed that ω DROID outperformed the baseline methods by detecting significantly more ω Bugs. As discussed in the previous section, ω DROID detected 30 true bugs. In contrast, baseline I failed to detect any WebView-related crash, while baseline II only detected one WebView-related crash for the app Shortyz [48]. The crash occurred after destroying and recreating activities. It is worth mentioning that the crash can also be detected by ω DROID since crashes also cause UI display inconsistencies. Such results indicate that injecting lifecycle events can help expose WebView-related crash yet such crashes are not common as we only observed one case in our evaluation. Specific oracles are thus needed to detect common types of ω Bugs. By combining *Lifecycle Event Injector* and *Lifecycle Misalignment Oracle*, ω DROID can detect ω Bugs effectively.

Answer to RQ5: ω DROID significantly outperformed the baseline methods, which confirms that our proposed oracles are useful in identifying anomalous app behaviors induced by common ω Bugs. Without effective oracles, injecting lifecycle events alone has limited effect on exposing ω Bugs.

7 DISCUSSIONS

7.1 Threads to Validity

Empirical study subject selection. Our empirical findings are obtained by studying ω Bugs in our selected Android apps. The findings are thus affected by the representativeness of these subjects. To mitigate this threat, we collected 124 ω Bugs from 51 open-source

android apps that are popular, large-scale, well-maintained, and diversified. Based on the empirical study, we proposed an automated ω Bugs testing technique ω DROID, which successfully detected 30 previously-unknown ω Bugs. This shows that our empirical findings can be generalized to other apps.

Keywords for ω Bug collection. When collecting code commits and bug reports related to ω Bugs, we only used general keywords “webview” and “javascript” while different projects may use different keywords to refer to WebView components. For example, in WordPress [64], WebViews are often called “posts” since WordPress uses WebViews to display blog posts. However, such app-specific keywords are hard to collect. Using them as search keywords can also lead to many irrelevant results. Therefore, we only used two general keywords, which already helped collect a considerable number of ω Bugs.

Errors in manual inspections. Our manual inspection may be subject to errors. To reduce human mistakes, four co-authors independently performed manual inspections and cross-checked each other’s results for consistency.

7.2 Limitations of ω DROID

Limitation of Test Event Generator. ω DROID leverages existing techniques to test apps containing WebViews. No existing work targets at generating test inputs to reach and thoroughly exercise all WebViews contained in an app. As a result, there is no guarantee that the test cases generated by ω DROID can reach all WebViews in an app and fully exercise their functionalities. This may leave some ω Bugs undetected. In future, we plan to improve the test case generation module of ω DROID to achieve more thorough testing.

Limitation of WebView UI model. ω DROID builds the UI model of a webpage based on the information given by UiAutomation [57]. Our UI models are thus subject to imprecisions caused by the limitations of UiAutomation (e.g., UiAutomation may fail to precisely model complicated webpages). In future, we also plan to explore alternative ways to define the UI model and improve the *Lifecycle Misalignment Oracle* module in ω DROID.

Handling multiple WebView instances. If one screen contains multiple WebView instances, ω DROID assumes that their order remains unchanged during the testing process. This may induce false positives if the order of the WebView instances changes after ω DROID injects lifecycle events. However, in our experiments, we did not observe any false positives caused by this limitation.

8 RELATED WORK

8.1 WebView Security

WebViews have posed various threats to an app’s security. Attacks on WebViews in Android system were first studied by Luo et al. [81, 82]. Since then, researchers have found more and more types of attacks on WebViews and proposed various defending techniques. For example, to prevent user private data from being leaked through JavaScript Interface, many techniques have been proposed to aid the detection and monitoring of cross-language information flow. Typical ones are BavelView [91], Spartan Jester [93], and Hybridroid [77]. Existing work [68] has also shown that malicious JavaScript code can invoke native methods and gain access to local resources such as files through JavaScript Interface. Most recently,

Li et al. [78] uncovered Cross-App WebView Infection (XAWI), a new attack that allows a remote adversary to spread malicious web content across different apps’ WebView instances and acquire stealthy control of these apps. Our work differs from these existing studies since we do not limit our scope in security issues but conduct a more comprehensive study on real-world ω Bugs.

8.2 Android Lifecycle Errors

Android app developers can easily make mistakes in managing app component lifecycles. To help improve app quality, Zaeem et al. [100] proposed a model-based testing technique QUANTUM to test user-interaction features of Android apps. QUANTUM generates test oracles based on the common sense expectations of app responses against user interactions. QUANTUM requires users to manually build an activity transition model for an app under test and it leverages trivial oracles such as screenshot differences for finding bugs. Our technique ω DROID differs from QUANTUM as it is fully automated and leverages the comparison of tree-based UI models for bug detection. Adamsen et al. [65] proposed THOR, which injects neutral event sequences (including lifecycle event sequences) into existing test cases based on the intuition that neutral event sequences should not affect the outcome of a test case. However, THOR relies on manually-written assertions to identify bugs. This is different from ω DROID, which is equipped with automated oracles. Shan et al. [94] proposed KREfinder to statically detect data lost when lifecycle events are triggered. This approach may not be applicable to WebView because the “data” of WebView usually reside in the loaded webpages but their approach does not support across-language data flow analysis.

9 CONCLUSION AND FUTURE WORK

In this paper, we conducted an empirical study on 124 ω Bugs collected from popular open-source Android apps. Our study identified the common root causes and consequences of ω Bugs, and investigated different factors that are needed to manifest ω Bugs. Our empirical study provides important insights that can guide future research on ω Bug testing and diagnosis. Based on our empirical findings, we proposed an automated testing technique ω DROID to detect common types of ω Bugs in Android apps. The evaluation of ω DROID on 146 subjects showed that ω DROID can effectively detect previously-unknown ω Bugs in real-world Android apps that are of developers’ concern. In future, we plan to collect more ω Bugs to perform a larger-scale empirical study. We also plan to study how to effectively generate user/system events and appropriate web content to thoroughly exercise WebView instances and leverage differential testing to expose more ω Bugs.

ACKNOWLEDGMENTS

The authors thank ASE 2018 reviewers and HKUST CASTLE group members for their constructive feedback. This work is supported by the Hong Kong RGC/GRF grant 16202917, MSRA collaborative research fund, Hong Kong PhD Fellowship Scheme, and the Nvidia academic program. The authors also would like to thank the Southern University of Science and Technology for the generous support on the research and travel.

REFERENCES

- [1] 2009. WebKit GTK on (electronic) paper. <https://people.collabora.com/~bari/slides/webkit-irex-fosdem-2009.pdf>.
- [2] 2018. 2048. <https://github.com/uberspot/2048-android>.
- [3] 2018. Aard Dictionary. <https://github.com/aarddict/android/>.
- [4] 2018. Activity Lifecycle. <https://developer.android.com/guide/components/activities/activity-lifecycle.html>.
- [5] 2018. Android Dashboards. <https://developer.android.com/about/dashboards/index.html>.
- [6] 2018. Android Lollipop. <https://developer.android.com/about/versions/lollipop.html>.
- [7] 2018. Android Monkey. <https://developer.android.com/studio/test/monkey.html>.
- [8] 2018. AnkiDroid. <https://github.com/ankidroid/Anki-Android/>.
- [9] 2018. AntennaPod. <https://github.com/AntennaPod/AntennaPod/>.
- [10] 2018. arXiv mobile. <https://github.com/jdeslip/arxiv-mobile>.
- [11] 2018. Barcode Scanner. <https://github.com/zxing/zxing>.
- [12] 2018. Budget Watch. <https://github.com/brarcher/budget-watch>.
- [13] 2018. Building Web Apps in WebView. <https://developer.android.com/guide/webapps/webview.html>.
- [14] 2018. Cherry. <https://github.com/gsantner/cherrymusic-android>.
- [15] 2018. Crossword. <https://github.com/billthefarmer/crossword>.
- [16] 2018. dandelion*. <https://github.com/Diaspora-for-Android/dandelion>.
- [17] 2018. Diary. <https://github.com/billthefarmer/diary>.
- [18] 2018. Drinks. <https://github.com/amasciul/Drinks>.
- [19] 2018. EteSync. <https://github.com/etesync/android>.
- [20] 2018. F-Droid. <https://f-droid.org/en/>.
- [21] 2018. Forecastie. <https://github.com/martykan/forecastie>.
- [22] 2018. ForRunners. <https://github.com/brvier/ForRunners>.
- [23] 2018. Freifunk Auto Connect. <https://github.com/WIStudent/FreifunkAutoConnectApp>.
- [24] 2018. Gadgetbridge. <https://github.com/Freeyourgadget/Gadgetbridge/>.
- [25] 2018. Gift Card Guard. <https://github.com/brarcher/gift-card-guard>.
- [26] 2018. GitHub. <https://github.com>.
- [27] 2018. Google Play. <https://play.google.com/store/apps>.
- [28] 2018. Hybrid apps more vulnerable. <https://www.mobileworldlive.com/apps/focus-apps/hybrid-apps-more-vulnerable-security-specialist-says/>.
- [29] 2018. Kimai. <https://github.com/gsantner/kimai-android>.
- [30] 2018. LinCal. <https://github.com/felixwienmuth/LinCal>.
- [31] 2018. Migrating to WebView in Android 4.4. <https://developer.android.com/guide/webapps/migrating.html>.
- [32] 2018. MTG Familiar. <https://github.com/AEFeinstein/mtg-familiar>.
- [33] 2018. Native, Web or Hybrid Apps? What's The Difference? <https://www.mobiloud.com/blog/native-web-or-hybrid-apps/>.
- [34] 2018. Nextcloud News Reader. <https://github.com/nextcloud/news-android/>.
- [35] 2018. OCRReader. <https://github.com/schaal/ocreader/>.
- [36] 2018. OpenVPN for Android. <https://github.com/schwabe/ics-openvpn>.
- [37] 2018. Padland. <https://github.com/mikifus/padland>.
- [38] 2018. PocketHub. <https://github.com/pockethub/PocketHub/>.
- [39] 2018. Polar Clock. <https://github.com/beriaai/PolarClock>.
- [40] 2018. RasPi Check. <https://github.com/eidotermihi/rpcheck>.
- [41] 2018. Reddinator. <https://github.com/micwallace/reddinator>.
- [42] 2018. RedReader. <https://github.com/QuantumBadger/RedReader/>.
- [43] 2018. Remove: Top frame navigations to data URLs. <https://bugs.chromium.org/p/chromium/issues/detail?id=684011&desc=2>.
- [44] 2018. Rental Calc. <https://github.com/brarcher/rental-calc>.
- [45] 2018. Save For Offline. <https://github.com/JonasCz/save-for-offline>.
- [46] 2018. Serval Mesh. <https://github.com/servalproject/batphone>.
- [47] 2018. sg. <https://github.com/SteamGifts/SteamGifts>.
- [48] 2018. Shortyz. <https://github.com/kebernet/shortyz>.
- [49] 2018. Slide. <https://github.com/ccrama/Slide/>.
- [50] 2018. SMS Backup+. <https://github.com/jberkel/sms-backup-plus>.
- [51] 2018. Snapcast. <https://github.com/badaix/snapcast>.
- [52] 2018. Stack Overflow. <https://stackoverflow.com>.
- [53] 2018. State of Application Development: 2017 Research Report. <https://www.outsystems.com/1/state-app-development-trends/>.
- [54] 2018. Termux. <https://github.com/termux/termux-app>.
- [55] 2018. ωDROID Project Website. <https://home.cse.ust.hk/~jhuao/wDroid.html>.
- [56] 2018. Twidere. <https://github.com/TwidereProject/Twidere-Android>.
- [57] 2018. UiAutomation. <https://developer.android.com/reference/android/app/UiAutomation.html>.
- [58] 2018. Vanilla Music. <https://github.com/vanilla-music/vanilla>.
- [59] 2018. Vespucci. <https://github.com/MarcusWolschon/osmeditor4android>.
- [60] 2018. Web Opac. <https://github.com/opacapp/opacclient>.
- [61] 2018. WebSettings. <https://developer.android.com/reference/android/webkit/WebSettings.html>.
- [62] 2018. WebView. <https://developer.android.com/reference/android/webkit/WebView.html>.
- [63] 2018. Wikipedia-Issue Tracker. <https://phabricator.wikimedia.org>.
- [64] 2018. WordPress. <https://github.com/wordpress-mobile/WordPress-Android/>.
- [65] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, 83–93.
- [66] Tanzirul Azim and Iulian Neamtii. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA 2013)*, Vol. 48. ACM, 641–660.
- [67] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, 238–249.
- [68] Erika Chin and David Wagner. 2013. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications*. Springer, 138–159.
- [69] Wontae Choi, George Necla, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA 2013)*, Vol. 48. ACM, 623–640.
- [70] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. IEEE, 429–440.
- [71] 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. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE 2018)*.
- [72] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE, 308–318.
- [73] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys 2014)*. ACM, 204–217.
- [74] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*. ACM.
- [75] Reyhaneh Jabbarvand and Sam Malek. 2017. μ Droid: an energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*. ACM, 208–219.
- [76] Casper S Jensen, Mukul R Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, 67–77.
- [77] SungHo Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybridDroid: static analysis framework for Android hybrid applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. IEEE, 250–261.
- [78] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. 2017. Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*. ACM, 829–844.
- [79] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 1013–1024.
- [80] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 396–409.
- [81] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC 2011)*. ACM, 343–352.
- [82] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2012. Touch-jacking attacks on web in android, ios, and windows phone. In *International Symposium on Foundations and Practice of Security*. Springer, 227–243.
- [83] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE 2013)*. ACM, 224–234.
- [84] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 599–609.
- [85] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, 94–105.

- [86] Atif M Memon, Martha E Pollack, and Mary Lou Soffa. 2000. Automated test oracles for GUIs. In *ACM SIGSOFT Software Engineering Notes*, Vol. 25. ACM, 30–39.
- [87] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST 2015)*.
- [88] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An empirical study of client-side JavaScript bugs. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2013)*. IEEE, 55–64.
- [89] Frolin S Ocariza Jr, Karthik Pattabiraman, and Benjamin Zorn. 2011. JavaScript errors in the wild: An empirical study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE 2011)*. IEEE, 100–109.
- [90] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. 2017. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. IEEE, 106–117.
- [91] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. 2017. BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. *arXiv preprint arXiv:1709.05690* (2017).
- [92] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 61–72.
- [93] Julian Sexton, Andrey Chudnov, and David A Naumann. 2017. Spartan Jester: end-to-end information flow control for hybrid Android applications. In *2017 IEEE Security and Privacy Workshops (SPW 2017)*. IEEE, 157–162.
- [94] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtii. 2016. Finding resume and restart errors in android applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, Vol. 51. ACM, 864–880.
- [95] Wei Song, Qingqing Huang, and Jeff Huang. 2018. Understanding JavaScript Vulnerabilities in Large Real-World Android Applications. *IEEE Transactions on Dependable and Secure Computing* (2018).
- [96] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*. ACM, 245–256.
- [97] Daniel R Thomas, Alastair R Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. 2015. The lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface. In *Cambridge International Workshop on Security Protocols*. Springer, 126–138.
- [98] 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 (ASE 2016)*. ACM, 226–237.
- [99] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2017. OASIS: prioritizing static analysis warnings for Android apps based on app user reviews. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*. ACM, 672–682.
- [100] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. 2014. Automated generation of oracles for testing user-interaction features of mobile apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST 2014)*. IEEE, 183–192.