RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps *

Umar Farooq
University of California, Riverside
ufaro001@ucr.edu

Zhijia Zhao
University of California, Riverside
zhijia@cs.ucr.edu

ABSTRACT

Portable devices, like smartphones and tablets, are often subject to runtime configuration changes, such as screen orientation changes, screen resizing, keyboard attachments, and language switching. When handled improperly, such simple changes can cause serious runtime issues, from data loss to app crashes.

This work presents, to our best knowledge, the first formative study on runtime change handling with 3,567 Android apps. The study not only reveals the current landscape of runtime change handling, but also points out a common cause of various runtime change issues – *activity restarting*. On one hand, the restarting facilitates the resource reloading for the new configuration. On the other hand, it may slow down the app, and more critically, it requires developers to manually preserve a set of data in order to recover the user interaction state after restarting.

Based on the findings of this study, this work further introduces a restarting-free runtime change handling solution – RuntimeDroid. RuntimeDroid can completely avoid the activity restarting, at the same time, ensure proper resource updating with user input data preserved. These are achieved with two key components: an online resource loading module, called HotR and a novel UI components migration technique. The former enables proper resources loading while the activity is still live. The latter ensures that prior user changes are carefully preserved during runtime changes.

For practical use, this work proposes two implementations of RuntimeDroid: an IDE plugin and an auto-patching tool. The former allows developers to easily adopt restarting-free runtime change handling during the app developing; The latter can patch released app packages without source code. Finally, evaluation with a set of 72 apps shows that RuntimeDroid successfully fixed all the 197 reported runtime change issues, meanwhile reducing the runtime change handling delays by 9.5X on average.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing systems and tools**; • **Software and its engineering**;

*Github homepage: https://github.com/ufarooq/RuntimeDroid.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MobiSys '18, June 10–15, 2018, Munich, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5720-3/18/06...$15.00
https://doi.org/10.1145/3210240.3210327

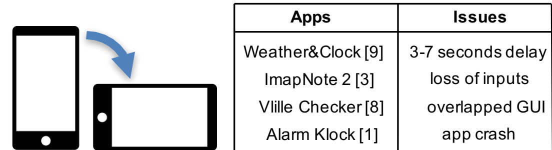

Figure 1: An example runtime change and its issues.

KEYWORDS

Runtime Configuration Change, Android, Event Handling

ACM Reference Format:
Umar Farooq and Zhijia Zhao. 2018. RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps . In *MobiSys '18: The 16th Annual International Conference on Mobile Systems, Applications, and Services, June 10–15, 2018, Munich, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3210240.3210327

1 INTRODUCTION

Nowadays, smartphone, tablets, and wearable devices are emerging as an essential component of modern life. According to IDC [27], over 1.46 billion smartphones were shipped in 2017. Among them, 85% are based on the Android platform.

Unlike traditional computers, such as desktops and laptops, these smart devices are more portable and subject to higher frequency of configuration changes, such as screen rotation, screen resizing, keyboard attachment, and languages switching. Such changes can happen at runtime while users interact with the devices, known as *runtime configuration changes* or *runtime changes*. Recent studies have shown that runtime changes happen regularly as users operate their apps. For example, on average, users change the orientation of their devices every 5 mins accumulatively over sessions of the same app [53]. For multilingual users, changing the language setting is often needed [16] and for tablet users, attaching an external keyboard often ease the uses of tablets [7]. As newer versions of Android system with multi-window supports getting adopted, it is projected that runtime changes will happen more frequently in more apps in the future. Each time a user drags the boundary between two split windows, a runtime change would be triggered [4].

Rise of Runtime Change Issues. Just as runtime changes become common to mobile apps, issues with runtime change mishandling also increases. Our preliminary examination of 765 repositories from Github shows that 342 of them had at least one issue due to runtime change mishandling, such as slowness, losing inputs, malfunctioning user interface, and even app crashes. All these issues can be triggered by simply rotating the device or attaching a keyboard. Figure 1 lists four example issues [1, 3, 8, 9] triggered each time a runtime change happens. In general, the runtime change issues can manifest in a variety of ways (see Section 3).

Formative Study. To better understand the landscape of runtime change handling and examine the root causes to various runtime change issues, this work presents, to our best knowledge, the first formative study on runtime change handling strategies and their related issues. The study is based on a large corpus of 3,567 Android apps with 16,160 activities and a focused corpus of 72 apps with 197 reported runtime change issues. All the studied apps are selected from Github based on their popularities and qualities, including many popular apps from Google Play Store (see Section 3).

The study results show that a large portion of Android apps (92.3%) rely on the passive *restarting-based* runtime change handling. Basically, the system first deconstructs the current user *activity*, including destroying all UI components and the internal logic data, then reconstructs the activity with the alternative resources (e.g., layouts and drawables) that can match to the new configuration (e.g., landscape orientation).

Though activity restarting facilitates the loading of alternative resources, the study results indicate that it raises risks of a series of critical runtime issues. First of all, restarting an activity invokes a sequence of callbacks (known as *lifecycle methods*), which may carry expensive operations, such as network connecting, database accessing, and other blocking operations. As a result, the app may become less responsive during runtime changes. More critically, to recover the user interaction state after activity restarting, it often requires developers to manually preserve a set of critical data. However, identifying such data and properly saving and restoring it are non-trivial and error-prone, especially as the complexity of app logic grows. When such data is not handled properly, runtime change issues as aforementioned would appear.

State of The Art. Some recent work [57] tries to identify the proper set of the data to save and restore during an activity restarting. However, since such data highly depends on the app logic, a generic data analysis often fails to identify the proper set of data. As a consequence, such approaches often produce over-conservative results – saving and restoring data that is not necessarily needed. Even worse, it is actually more challenging to verify if the data is correctly saved and restored, due to the availability of a wide range of APIs used for data saving and restoring [10–13].

Solution of This Work. Unlike prior efforts, this work proposes a *restarting-free* runtime change handling solution – RuntimeDroid. By preventing the activity from restarting, RuntimeDroid ensures that all the activity data remains live after runtime changes, thus making the data preservation a trivial task.

On the other hand, without activity restarting, the resources needed for the new configuration will not be automatically loaded. To address this issue, RuntimeDroid features HotR – a novel *online* resource loading solution that systematically loads the resources needed for the new configuration while the activity remains live. In cases where new UI resources are loaded, it will automatically migrate the properties of the original UI components to the newly generated UI components. We refer to this data migration technique as *dynamic view hierarchy migration*.

For easy adoption of RuntimeDroid, this work presents two alternative implementations:

- RuntimeDroid-Plugin: an Android Studio plugin that allows developers to easily adopt the restarting-free runtime change handling into the current app development by simply extending a customized activity class.
- RuntimeDroid-Patch: an automatic patching tool that can patch a compiled Android app package (i.e., APK file) to enable restarting-free runtime change handling, without source code.

In neither implementation would RuntimeDroid require any modifications to the existing Android framework.

We evaluated RuntimeDroid on a corpus of 72 Android apps from Github and Google Play Store with 197 reported runtime change issues. The results show that, after applying RuntimeDroid to the apps with runtime change issues, 197/197 issues have been fixed, thanks to the adoption of restarting-free handling strategy. Furthermore, RuntimeDroid reduces the runtime change handling time by 9.5X on average. On the other hand, RuntimeDroid may introduce some space overhead due to the factoring or patching, but typically less than 1% after packaging.

Contributions. This work makes a four-fold contribution.

- It provides, as far as we know, the first formative study on the landscape of runtime change handling, and points out a type of emerging issues in mobile apps – *runtime change issues* and its root cause – *activity restarting*.
- It proposes a versatile restarting-free runtime change handling solution – RuntimeDroid, which mainly consists of two novel components, an online resource loading module and a dynamic view hierarchy migration technique.
- It offers two practical implementations: RuntimeDroid-Plugin and RuntimeDroid-Patch. They together make the adoption of restarting-free runtime change handling an easy task both during and after the app development.
- Finally, this work evaluates RuntimeDroid and demonstrates its capability in addressing real-world runtime change issues and improving the responsiveness in general.

In the following sections, we will first give a brief introducation of Android programming and runtime change handling (Section 2), then present the formative study on runtime change handling, including its common issues (Section 3). After that, we will present RuntimeDroid and its implementations (Section 4), followed by the evaluation (Section 5). Finally, we will discuss related work and conclude this work (Sections 6 and 7).

2 BACKGROUND

In this section, we first briefly introduce the programming model for Android apps, then discuss runtime changes and their basic handling strategies.

2.1 App Programming Model

Android apps are primarily written in Java and built around the concept of *activities*. In brief, an activity represents a screen with UI components and the app logic behind the screen.

Activity Lifecycle. As the user interacts with an app, an activity may go through a series of stages in its lifecycle, such as `created`, `started`, `resumed`, `paused`, `stopped` and `destoryed`. To help the app transition among the stages, Android system provides a core set of lifecycle callback methods that will be invoked during the transitions, as illustrated in Figure 2. By overriding the lifecycle

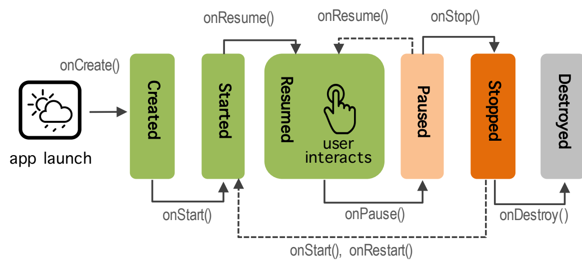

Figure 2: Lifecycle methods of an activity

callbacks, developers can customize the responses to lifecycle stage transitions, such as establishing server connections, initializing the data structures, or acquiring system resources (e.g., camera).

Event-driven Model. Like other GUI frameworks, Android models the user-app interactions as a sequence of event handling. Under the Android system, typical events include user input events (e.g., clicks and swipes) and sensor events (e.g., GPS and orientation changes). To respond to these events, developers need to implement corresponding handler methods. For example, to handle long-touch clicks of a button (i.e., holding the button for one second), developers need to first register a long-touch click listener for the button, then override the `onLongClick()` handler method.

To process events, Android adopts a single-thread model. When an app is started, a Linux process is created with a single thread, called the *UI thread*. The UI thread receives event messages and dispatches them to the corresponding callback/handler methods to respond to the events. This single-thread model requires developers to limit the workload of UI thread to keep the app responsive.

2.2 Runtime Change Handling

Unlike the traditional operating systems for desktops and laptops, Android is an operating system targeting the mobile devices that may frequently encounter various *runtime configuration changes* during its interactions with users.

Runtime Changes. Table 1 lists the runtime configuration changes defined by the Android API (Level 25). There are several runtime changes related to the screen, including screen size change, screen orientation change and touch screen change. Note that a common device rotation will trigger both screen orientation and screen size changes (since Android 3.2) and window resizing in multi-window mode will only trigger screen size changes (since Android 7).

Besides screen-related changes, there are also runtime changes for cellular network, keyboard availability, language, font size, and layout direction. Among these changes, screen orientation change, screen size change, and keyboard availability change are commonly considered by developers (see Section 3).

Resources. During a runtime change, an app may need to load alternative *resources* based on the new configuration. For example, when switching the screen from portrait mode to landscape mode, an app may need to load a different layout designed for landscape mode. In general, Android allows developers to provide *alternative resources* for different configurations to enable rich user experiences. They can also specify the *default resources* in cases no resources are available for the new configuration. All app resources are grouped and placed in folder `/res` under the project root directory.

Table 1: Runtime changes (API 25)

Change	Description
`mcc/mnc`	IMSI mobile country/network code
`locale`	language
`touchscreen`	touchscreen
`keyboard`	keyboard type
`keyboardHidden`	keyboard accessibility
`fontScale`	font scaling factor
`uiMode`	user interface mode
`orientation`	screen orientation
`screenSize`	available screen size
`smallestScreenSize`	physical screen size
`layoutDirection`	layoutDirection

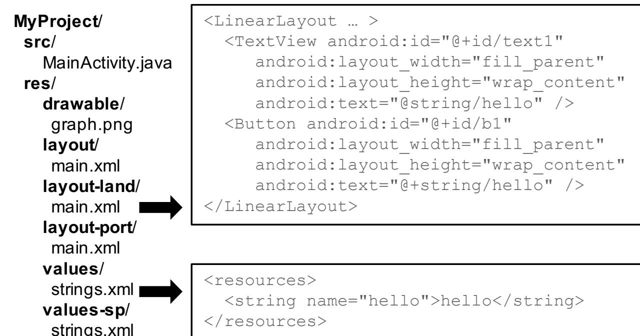

Figure 3: Example resources

Figure 3 shows an example resource folder, where the sub-folder `/layout` contains the default layout in XML format `main.xml` for the main activity, while folders `/layout-land` and `/layout-port` contain the layouts customized for the landscape mode and portrait mode, respectively. In addition to layouts, some other commonly used resources include strings, drawable images, menus, colors, dimensions, and styles. Developers can also define resources for different combinations of configurations, such as resources for Spanish language under the landscape mode (`/layout-sp-land`). A complete list of available resources and their naming conventions can be found in the Android API Guides [5].

For easy access to the resources, Android dynamically generates a resource class `R.java` based on the available resources (i.e., `/res`). The generation may happen in two situations: (i) when an activity is created and (ii) when a runtime change happens. The second situation is critical to the design of RuntimeDroid (see Section 4).

To effectively handle various runtime changes and load needed resources accordingly, Android offers two basic strategies: *H1 - restarting-based handling* (default) and *H2 - customized handling*.

H1*: Restarting-based Handling*. By default, to handle a runtime change, Android would first kill (deconstruct) the current activity, then restart a new activity with resources matched to the new configuration. This process typically involves transitions of all the lifecycle stages of an activity, from `Paused` all the way back to `Resumed` again (see Figure 2).

In the simplest cases, the default runtime change handling does not require any extra programming efforts from the developers, except providing alternative resources for certain configurations based on the design of their app. Beyond that, the activity restarting automatically handles and updates the resources accordingly.

However, in slightly more complex cases, an activity may carry various data, such as articles in a news app or the state of a game. When the old activity gets killed during a runtime change, such data is destroyed together. To avoid content reloading or losing the user interaction state, developers need to preserve a set of critical data during the activity restarting. There are two basic ways to preserve the data: *saving/restoring activity state* and *retaining objects*. For easy references, we refer to them as H1.1 and H1.2, respectively.

- **H1.1:** *saving/restoring activity state*. Before deconstructing the activity, the system first calls `onSaveInstanceState()` method to save the activity state into a `Bundle` object as a collection of key-value pairs. In fact, by default, this method already saves some transient information for some UI components, such as the text in an `EditText` or the selection of a `CheckBox`. To save additional state data, developers need to override this method and add extra key-value pairs into the `Bundle` object.
  When an activity instance is recreated, developers can recover the activity state by extracting data from the `Bundle` object, which might be performed either in `onCreate()` callback or in `onRestoreInstanceState()` callback.
- **H1.2:** *retaining objects*. When the data to save is complex or substantial in size, a more suitable way is retaining the data as objects. This can be achieved with *fragment*, which represents a behavior or a portion of an activity. There are four basic steps: (i) extend the `Fragment` class and declare references to the stateful objects; (ii) call `setRetainInstance(true)` when the fragment is firstly created; (iii) attach the fragment to the activity; and (iv) retrieve the fragment with `FragmentManager` when the activity is restarted. Despite the activity restarting, the data encapsulated by fragments remains accessible.
  Besides *fragment*, another solution to retain objects during activity restarting is adopting the `ViewModel` and `LiveData` components, which are recently introduced in the Architecture Components Library [19]. In this case, developers need to create a `ViewModel` class with critical data encapsulated, where the critical data is declared as `LiveData`. In this way, the Android framework will retain the `ViewModel` object while the activity object is recreated (i.e., activity restarting), hence the critical data will remain live after the restarting.

Both H1.1 and H1.2 can be adopted by the default runtime change handling for preserving critical data. However, the complexity lies in identifying the various data to preserve. As shown later in the formative study, developers often fail to identify the critical data or do not save/restore it correctly.

H2: Customized Handling. Instead of letting the activity to restart, developers may choose to directly program the runtime change handling. To do this, developers first need to set the runtime change flag `android:configChanges` for self-handling changes in the app configuration file (i.e., `Manifest.xml`). Once flagged, a runtime change will no longer result in any activity restarting. Instead, it will trigger `onConfigurationChanged()` callback. By overriding this callback, developers can manually load the altenative resources for the new configuration.

However, manually updating resources for different runtime changes requires deep understanding on the types of resources and their allocation mechanisms, thus this option is usually beyond the reach of most Android developers. As shown later in the formative study, few apps (7.7%) actually adopt this option in practice.

For certain runtime changes (e.g., screen orientation and screen size), developers may opt to disable them by setting flags in an activity declaration (since API 24). Once they are specified, users cannot change the orientation or resize the screen (in multi-window mode) under the activity. It is obvious that this setting limits app functionalities, thus may negatively affect the user experience. We refer to this option as **H3**. Note that H3 is only available to a subset of runtime changes, rather than a general runtime change handling.

To understand how different runtime change handling strategies (H1-H3) are adopted in practice, we next present a formative study on real-world Android apps.

3 FORMATIVE STUDY

Our formative study on runtime change handling aims to address two fundamental questions:

- **RQ1 (Landscape):** *How do developers program runtime change handling? What are the common practices?*
- **RQ2 (Common Issues & Causes):** *What are the basic types of runtime change issues? Are there any common causes?*

For each question, we first present the corpus and methodology, then discuss the results and implications.

3.1 RQ1: Landscape

First, we examine the common practices of runtime change handling in real-world Android apps.

Corpus-L. We collected Android apps from Github [2], mainly for two reasons. First, as the largest code hosting service provider, Github hosts a large number of industrial-grade Android apps, such as Telegram messenger [28], K-9 email [23], Google I/O [21], Amaze File Manager [25], Timber player [26], and Wordpress [31], just to name a few. Many of these apps are hosted on Google Play Store [22]. Second, with the availability of source code, our formative study provides more precise runtime handling analysis.

To focus on popular apps, we sorted the searching results based on the number of stars and the number of forks of each repository. That is, only the top Android apps on Github are selected. To ensure the collected repositories are Android apps, we checked the source code of every selected project to ensure the existence of an app manifest file `AndroidManifest.xml` (required by Android). After searching and filtering, the corpus contains 3,567 apps with 16,160 activities and 24.7 M lines of code, referred to as *Corpus-L*.

Methodology. To analyze the runtime change handling for the large volume of apps in *Corpus-L* efficiently, we developed an automatic code analysis tool – RuntimeAnalyzer.

For each app in the corpus, RuntimeAnalyzer first parses its manifest file and collects the basic runtime change configurations for each registered activity. These include the settings for screen orientation changes (`screenOrientation`) and resizing changes (`resizableActivity`), and the setting for self-handling runtime changes (`configChanges`). If `configChanges` is set (i.e., H2), the analyzer will parse the activity class to examine if the callback `onConfigurationChanged()` has been overridden.

To better understand the data preserving methods in the default handling (Section 2), RuntimeAnalyzer also checks the uses of

Table 2: Uses of Runtime Change Handling.

Handling Strategies	#activities	#app
Activity Restarting	14,934 (92.4%)	3,293 (92.3%)
Customized Handling	1,226 (7.6%)	274 (7.7%)

Table 3: Uses of Restarting-based Handling (H1).

Data Preservation Method	#activities	#app
`saveInstanceState` (H1.1)	999 (7.6%)	458 (12.7%)
Object Retaining (H1.2)	223 (1.7%)	105 (3.0%)
No Data Preservation	11,792 (90.6%)	3,024 (84.3%)

Table 4: Uses of Customized Handling (H2).

Overridden Callback	#activities	#app
`onConfigurationChanged`	155 (12.6%)	87 (31.7%)
No Overriding	1,071 (87.4%)	187 (68.3%)

state saving callback `saveInstanceState()` and the `Bundle` object. If `saveInstanceState()` is overridden and the `Bundle` object is also unpacked either in `onCreate()` or `restoreInstanceState()`, then the handling is H1.1. Similarly, if a fragment is attached to the activity with a call to `setRetainInstance(true)` or a `ViewModel` is declared, then the handling would be categorized as H1.2.

Results. Tables 2-4 report the statistical results of the study. As shown in Table 2, the restarting-based runtime change handling (H1) absolutely dominates the handling strategies. Among the 16,160 activities examined, 92.4% choose H1, which covers 92.3% of total apps. This is mainly due to its lower barriers to programming than the customized handling (H2). H2 requires solid understanding of resource loading mechanisms (see Section 2).

Among the activities with restarting-based handling, only 13.9% leverage the callback `saveInstanceState()` (H1.1) to preserve the data and 15.4% adopt object retaining (H1.2). In contrast, a large portion of the activities (68.3%) provide no mechanisms for data preserving at all. As the activity restarting invokes lifecycle methods, such as `onCreate()` and `onStart()`, which provide the basic UI initialization and even the data loading, in many cases, the screen may appear the same as the one before the runtime change, especially for simple activities. However, as the logic complexity of an activity grows, restarting the activity without sufficient data preservation makes the app vulnerable to various runtime change issues (see Section 3.2).

Among the activities that choose the customized handling (H2), only about one third (31.7%) actually override the callback method `onConfigurationChanged()`. For the other two thirds, developers do not provide alternative resources for different configurations, thus there would be no need to override the callback. These results indicates that manually resources updating is only practiced in a limited way, due to its complexity.

Table 5 lists the statistics of runtime changes that are flagged for self-handling (i.e., listed in `configChanges`). Among them, the most popular ones include `orientation`, `keyboardHidden`, and `screenSize`. Ironically, despite the flagging, as just mentioned in Table 4, very few activities actually implement the "self-handling". They simply use the flags to prevent activities from restarting under certain runtime changes.

Another interesting finding is a gap between `screenSize` (22.9%) and `orientation` (32%). Actually, to handle orientation changes,

Table 5: Uses of Configuration Changes Properties.

Changes	#activities	#app
keyboard	326 (8.8%)	69 (25.2%)
mnc	17 (0.5%)	12 (4.4%)
mcc	16 (0.4%)	11 (4.0%)
locale	71 (1.9%)	20 (7.3%)
navigation	29 (0.8%)	18 (6.6%)
fontScale	22 (0.6%)	10 (3.6%)
layoutDirection	10 (0.3%)	6 (2.2%)
keyboardHidden	986 (26.8%)	225 (82.1%)
orientation	1,178 (32.0%)	271 (98.9%)
screenLayout	129 (3.5%)	20 (7.3%)
uiMode	20 (0.5%)	15 (5.5%)
screenSize	844 (22.9%)	198 (72.3%)
smallestScreenSize	37 (1.0%)	10 (3.6%)

developers need to specify both `screenSize` and `orientation` (since Android 3.2). This gap implies that there exist many misuses of the runtime change configurations `configChanges`.

Finally, the study shows that a small ratio of activities (15.5%) are set with a fixed orientation (either landscape or portrait) and only 4.3% apps have fixed orientation for all the activities. Moreover, it shows no activities actually disable the resizing in multi-window mode. The results indicate that for most apps, developers would not like to limit the functionalities by disabling the runtime changes.

3.2 RQ2: Common Issues and Causes

Corpus-S. To examine the issues in runtime change handling, we collected another corpus with apps actually suffering from runtime change issues, named as *Corpus-S*.

Corpus-S consists of 72 Android apps collected from Github, for the same reasons as *Corpus-L* (see Section 3.1). 36 out of the 72 apps are also hosted on Google Play Store [22], including quite a few highly popular ones, such as *Loop - Habit Tracker* [24] with 1M installs, *WiFiAnalyzer* [30] with 1M installs, Barcode Scanner [20] with over 100M installs, and among others. Another reason that we choose Github is for its availability of issue reports. The traceable issue reports on Github allows us to easily identify specific apps with runtime change issues. Together, *Corpus-S* composes of 507 activities with a total of 1.5M SLOC and 197 runtime change issues.

Methodology. We manually examine the runtime change issues one by one and categorize them based on their manifestation. Overall, there are four basic types: *T1 - poor responsiveness*, *T2 - lost state*, *T3 - malfunctioning UI*, and *T4 - app crash*. Together, they reflect the common issues that apps encounter during runtime changes.

T1: Poor Responsiveness. This type of issues causes significant delays during runtime changes. The app `Weather&Clock` shown in the introduction (Figure 1) falls into this category. In addition, the study found three other apps reported with unexpected delays during runtime changes.

Note that, poor responsiveness, despite often appearing with runtime change mishandling, is less likely to be reported. First, as a non-functional issue, some users and developers often choose not to report it as "an actual issue". Second, due to the lack of expertise in runtime change handling, some developers consider the issues as "how it is supposed to be" or "the issue of Android".

Causes: There are two basic conditions jointly contributing to the occurrences of poor responsiveness during runtime changes: (i)

the use of restarting-based handling H1 and (ii) the existence of blocking operations in the lifecycle callbacks.

The first condition causes an activity to restart, going through the whole sequence of lifecycle stages (see Figure 2). Meanwhile, the lifecycle methods are invoked one by one, from `onPause()` to `onResume()`. If any of these lifecycle methods performs some blocking operations, such as I/O operations, network connections or Bitmap manipulations, there will be substantial extra delay(s) for the runtime change handling. When the total delay becomes significant, the user would observe it. In the weather app example (`Weather&Clock`), the activity restarting triggers re-connecting to the server and re-downloading the map and weather data. Together, they contribute a delay of 3-7 seconds.

T2: Lost State. This is the most common type of issues triggered by runtime changes – *losing user interaction state*. Note that runtime changes may happen at any point during a user-app interaction session. At the time a runtime change occurs, the user may have already performed some actions and changed the state of some UI components, such as entering some text, selecting an item, or opening a dialog. With lost state issues, such user inputs will be lost during runtime changes. In more serious cases, the users may even lose their login state. Consequences like these frustrate users, undermining the overall impression of the app qualities.

Causes: The study shows that most lost state issues are due to the missing or insufficient data preservation with the use of restarting-based handling (H1). As the activity is restarted, the associated UI components will be destroyed together with their attributes, like text, selection, and position. For built-in UI components with assigned IDs, the system can automatically save/restore certain editable attributes (e.g., text in `EditText`). However, this may not cover all critical attributes of all UI components, not to mention the internal logic data. Furthermore, the study shows that despite the saving/restoring, the data may be reset with initial values during runtime changes (e.g., by the `onCreate()` callback). The results indicate that many developers who adopted the restarting-based handling (H1) are not prepared for such detailed data handling requirements, thus their apps may suffer from the loss of the user interaction state at runtime changes.

T3: Malfunctioning UI. In some use scenarios, the study shows that runtime changes can result in malfunctioning user interface, such as overlapped views, stretched images, and mispositioned menus. For example, in the setting view of `Vlille Checker` [8], an app for self-service biking, runtime changes result in two layers of GUIs overlapped with each other.

Causes: The issue happened in `Vlille Checker` is due to the misuse of fragments in restarting-based handling (H1). When a runtime change occurs, a new activity is restarted with a new fragment attached. Meanwhile, the old fragment is still retained by the system, thus overlapped with the new one. In general, the malfunctioning UI issues are often caused by the improper ad-hoc handling of UI components during the activity restarting.

T4: App Crash. In some cases, a simple runtime change, like screen rotation, can cause an app to crash. When it happens, a message "*Unfortunately your app has stopped*" pops to the screen. This class of issues is of the most severe type.

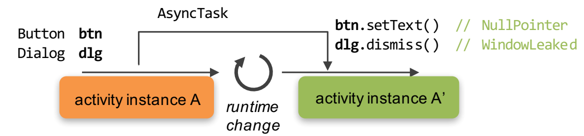

Figure 4: A common app crash scenario, caused by the misuse AsyncTask during activity restarting.

Causes: The study shows that a few Java and Android exceptions commonly contribute to the app crashes, including `NullPointer`, `WindowLeaked`, `IndexOutofBound`, and `Instantiation`. Among them, `NullPointer` and `WindowLeaked` are the most common ones, which are often triggered by the misuse of asynchronous function calls (e.g., `AsyncTask`) with the restarting-based runtime change handling H1. Figure 4 illustrates a common app crash scenario.

Before the runtime change, an `AsyncTask` instance was created by the activity instance A. Some time after the runtime change, the `AsyncTask` instance finished and attempted to update a UI component and dismiss a dialog. However, after the restarting, the activity instance had become A'. Thus, neither the dialog or the GUI component is available. Accesses to these components would result in a `NullPointer` exception and `WindowLeaked` exception, respectively, causing the app to crash.

Discussion on Common Causes. On one hand, runtime change issues exhibit a variety of consequences, from the loss of an input to brutal app crashes. On the other hand, based on the cause analysis, the study results indicate that *they share a common condition* – the adoption of the restarting-based runtime change handling (H1).

In general, activity restarting requires developers to take special care of the of lifecycle method design (T1), the GUI attributes (T2 and T3), the state of activity logic objects (T2), as well as use of asynchronous function calls (T4). These strong requirements make the runtime change handling a tedious and error-prone task.

Instead of trying to fulfill all the requirements as mentioned above, a clean solution is to *avoid the activity restarting (H1) during runtime changes*. This will remove a necessary condition for most runtime change issues (T1-T4). However, as mentioned earlier, the other strategies (H2 and H3) are either beyond the reach of developers or limit the functionalities of the apps. To bridge this gap, this work proposes RuntimeDroid – a restarting-free runtime change handling solution that can be easily adopted by developers.

4 RUNTIMEDROID

In this section, we introduce a restarting-free runtime handling solution – RuntimeDroid. We first describe its basic ideas, then elaborate its key components, which consists of an *online resource loading* module – HotR and a *dynamic view hierarchy migration* technique. Finally, we discuss two alternative implementations of RuntimeDroid for easy adoption.

4.1 Challenges

As mentioned earlier, to prevent activities from restarting during runtime changes, developers can set the `configChanges` flag in the activity configuration (i.e., customized handling H2). However, this requires developers to manually load resources for the new configuration, which, unfortunately, is very challenging for many Android developers, due to three reasons:

- *Complexity of Resource Types.* In the latest API (API 27) of Android, there are 16 types of resources for mobile apps, each requiring a specific loading mechanism.
- *Complexity of Resource Uses.* Resources can be statically bound to other resources, like layout, or dynamically referred in the callbacks of the activity class.
- *Dynamic Nature of UI Components.* As the user interacts with an activity, the properties of some UI components might be changed dynamically. Such changes need to be preserved while the resources are loaded. Even more complex, there might be UI components added or deleted during the user interaction.

The above three complexities involved in resource loading make the customized runtime change handling beyond the reach of most Android developers. To address this challenge, we next present an automatic online resource loading module – HotR. HotR is able to load resources for the new configuration while the current activity remains live. Moreover, it does not depend on the app logic.

4.2 HotR

The purpose of HotR is to load resources needed for the new configuration without restarting the activity. In the following, we first define the concept of *alternative resources*, then present the major components of HotR.

Depending on the design, an activity may have different versions of resources that are defined for different configurations. For easy references, we define *alternative resources* as follows.

Definition 4.1. During a runtime change, an alternative resource is a resource designed for the configuration after the runtime change, but not used by the configuration before the runtime change.

We now present the major components in HotR following the order that they are employed in actual resource loading.

When a runtime change occurs, HotR first examines the needs for resource loading. To achieve this, HotR constructs two *hashmaps* for recording resources used before and after the runtime change.

***C1: Resource Hashmap Construction*.** For a given configuration C, a *resource hashmap* (*RH*) contains an entry for each resource declared in C, except the layouts. Unlike a typical hashmap, the key in *RH* is the "content" of a resource, represented as a string, such as text "Enter" in a `string` resource. For non-string resources, such as `drawable` (e.g., bitmaps), `color`, or `dimension`, they will be either hashed or serialized into strings. The value in *RH* is the resource ID, which is uniquely assigned by the system. Both the "contents" and the IDs of resources for the current configuration can be accessed from the built-in class `R`.

For example, the following resources for the portrait mode will be compiled into the `R.class`, then built into the resource hashmaps RH_{port}. After the screen orientation, the process happens again to form the resource hashmaps RH_{land}.

/res/values-port/res.xml

```
<string name="enter">Enter</string>
<color name="yellow">0xffffff00</color>
```

/res/values-land/res.xml

```
<string name="enter">Enter Name</string>
<color name="yellow">0xffffff00</color>
```

RH_{port} :

Key	Value
"Enter"	134
"0xffffff00"	152

RH_{land} :

Key	Value
"Enter Name"	134
"0xffffff00"	152

Note that both "Enter" and "Enter Name" have the same resource ID (134). This is because they are given the same `name` ("enter") in the declarations, hence system yields the same ID for them.

As we will explain shortly, layouts play a special role in resource loading, thus HotR treats them separately.

***C2: Alternative Resource Identification*.** Given a runtime change, HotR determines the needs for resource loading by calculating the difference resource hashmap RH_{diff} between the old resource hashmap RH_{old} with the new one RH_{new} in terms of the key set.

$$RH_{\text{diff}} = RH_{\text{old}} - RH_{\text{new}} \tag{1}$$

If RH_{diff} is non-empty, then HotR would consider the existence of alternative resources, hence the needs for loading resources. Following the example in C1, the difference would be:

RH_{diff} :

Key	Value
"Enter"	134

Thus, there exists an alternative resource ("Enter Name") to load.

Note that the above process for determining resource loading may yield *false positives*, that is, there might be no actual needs for resource loading despite a non-empty RH_{diff}. This is because the `R` class consists of the resources for all the activities, not only for the current activity. Such false alarms could be avoided with static analysis or activity-level `R` class supports from Android runtime. In fact, even in the presence of false alarms, our evaluation shows that the online resource loading by HotR is still much faster than the restarting-based mechanism.

For the layout, HotR can obtain its resource ID for the current activity (via `setContentView(layoutID)`), hence it can directly compare the new layout L_{new} with the old one L_{old} to determine the existence of an alternative resource (i.e., $L_{\text{diff}} = L_{\text{old}} - L_{\text{new}}$). In this case, there will be no false positives.

If RH_{diff} and L_{diff} are both empty, HotR will skip the resource loading and terminate – the runtime change handling is completed.

***C3: Property-Resource Mapping*.** In Android apps, resources are mainly used for defining the view properties. For example, a string resource can be used as the text property of a `EditText`, a color resource can be the background of a `LinearLayout`, and a drawable can be linked to the resource of an `ImageView`.

Knowing the mapping between the view properties and their corresponding resources can help locate the uses of resources, hence facilitating the loading process. For this purpose, HotR constructs the property-resource mapping M_{PR} based on the programming conventions. For example, the text property of view `EditText` is mapped to the `string` resource.

M_{PR} :

View Property	Resource Type
`EditText.setText( )`	string

Note that the mapping M_{PR} can pre-constructed offline.

***C4: Resource Loading*.** All the components from C1 to C3 are the preparation for the actual resource loading – C4. When a runtime change occurs and the outcome of C2 is positive (i.e., at least one

of RH_{diff} and L_{diff} is non-empty), then HotR will initiate the actual resource loading process.

Note that during the user-app interactions, the properties of some views might be modified (e.g., text property of `EditText`). Moreover, some views might be even removed or added dynamically (by event handlers). HotR treats these cases differently.

For easy references, we categorize the views into two classes: *static views* and *dynamic views*, defined as follows.

Definition 4.2. For a given activity, its views that are declared in the layout file are referred to as *static views*. Correspondingly, the resource loading for static views is called *static resource loading*.

Definition 4.3. For a given activity, its views that are added or deleted at runtime are referred to as *dynamic views*. The resource loading for dynamic views are called *dynamic resource loading*.

Note that a view that was originally declared in the layout may be deleted at runtime. In this case, we say that a static view turns into a dynamic view after it is deleted. So whether a view is static or dynamic depends on when we refer to it.

Next, we first discuss the static resource loading. For dynamic resource loading, which is more complex, we leave it to Section 4.3 when we discuss the dynamic view hierarchy migration.

To perform static resource loading, HotR leverages a handy callback from the system `setContentView()`. Though the callback is used for loading layout resources, it actually also loads other types of resources *implicitly*. This is because the layout consists of all the static views (see Definition 4.2). After the layout is loaded, all the static views will also have their properties loaded with alternative resources, automatically ensured by the system.

However, note that the properties of static views may be updated at runtime by some callbacks. Thus, simply loading the alternative resources for static views may lead to inconsistent view properties. In addition, as mentioned earlier, some static views may turn into dynamic views (i.e., being deleted). We will address these issues with *dynamic view hierarchy migration*. Before that, let us first finish the last component of HotR.

C5: Resource Reference Updating. The last component of HotR is about the resources that are referred in the callbacks of the activity class. For example, the following statements access a string resource or a view resource from some callback:

```
String hello = getString(R.string.hello);
TextView name = findViewById(R.id.nameview);
```

When the alternative resources are loaded, we need to make sure that *the corresponding references point to the newly loaded resources, instead of the original ones.* We separate this discussion into two cases based on locations where the resources are referred:

- *Local Resource References.* When resources are referred locally in a callback method, the reference updating will be naturally ensured, thanks to the automatic updating of `R` class. When a runtime change happens, the system would automatically recompile the `R` class for the new configuration (see Section 2.2). Hence, when a callback is invoked after the runtime change, its resource references (through the `R` class) will automatically point to the alternative resources.
- *Global Resource References.* However, the same situation does not apply to the cases where resource references are declared globally (i.e., the activity class level). This is due to the fact that global resource references may not be *reassigned* after a runtime change. For example, `String str` is first declared at the activity level, then initialized in `onCreate()` with

  ```
  str = getString(R.string.hello);
  ```

  Even though the `R` class has been updated with references to the alternative resources, the assignment to `str` will not be invoked again without activity restarting, therefore it will remain pointing to the original resource.
  To address the *stale* global resource references issue, HotR performs a *resource reference localization* procedure for each global resource reference. Basically, given a global resource reference *p*, HotR first traces where *p* is used, then inserts an assignment with the corresponding reference in the `R` class right before *p* is used (e.g., `p=getString(R.string.hello)`). By adding the "assignment" for global resource references, this reference localization ensures the accesses to the correct references, just like accessing local resource references.

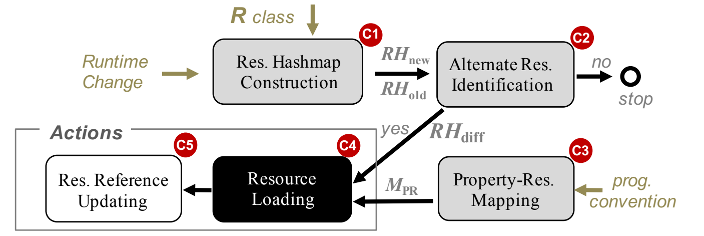

Figure 5: High-level Workflow of HotR.

Summary. Figure 5 summarizes the major components in HotR with a high-level workflow. Components C1 to C3 prepare for the actual resource loading C4. If no alternative resources are found in C2, HotR will stop after C2; otherwise, it will finish C4 and C5.

So far, we have introduced the static resource loading in C4. Next, we discuss the other part of C4 – dynamic resource loading, as well as how to address the view properties that are updated at runtime. We address the two problems together with a novel *dynamic view hierarchy migration* technique.

4.3 Dynamic View Hierarchy Migration

Before presenting the technique, we first introduce a couple core concepts that are used in our design. The UI components (i.e., views in Android's term) of an activity forms a hierarchical structure, typically with a type of layout view as the root (e.g., `LinearLayout`). Depending on when the view hierarchy is referred to, we define *static view hierarchy* and *dynamic view hierarchy* as follows.

Definition 4.4. A *static view hierarchy*, denoted as $\mathcal{V}$, is the initial view hierarchy that is derived from the activity layout (XML file).

Definition 4.5. A *dynamic view hierarchy*, denoted as $\widetilde{\mathcal{V}}$, is the view hierarchy that is referred to while the user interacts with the activity. $\widetilde{\mathcal{V}}$ may evolve over the interaction. When the layout of an activity is just loaded, $\widetilde{\mathcal{V}}$ equals to the static view hierarchy $\mathcal{V}$.

The static resource loading described in component C5 of HotR (see Section 4.2) essentially builds a static view hierarchy for the new configuration. However, during user interactions, some views'

Algorithm 1 Dynamic View Hierarchy Migration

```
1: 𝒱_old: the static view hierarchy before runtime change
2: 𝒱̃_old: the dynamic view hierarchy before runtime change
3: 𝒱_new: the static view hierarchy after runtime change
4: 𝒱̃_new: the dynamic view hierarchy after runtime change
5:
6: /* preparation */
7: V_dyn = compare(𝒱_old, 𝒱̃_old);                 ▷ identify dynamic views
8: V_static = derive(𝒱_old);                        ▷ derive static views
9: P_mut = find(Activity.class);                    ▷ identify mutable properties
10:
11: /* migrate static views */
12: for each view v in V_static do
13:     for each property p of v do
14:         if p ∈ P_mut then
15:             if p has RH_diff then                ▷ RH_diff: res. w/ alternatives
16:                 loadResource(p, RH_diff);
17:             copy(p, 𝒱_new(v));                   ▷ 𝒱_new(v) returns the view w/ same id
18:
19: /* migrate dynamic views */
20: for each view v in V_dyn do
21:     if v was deleted from 𝒱_old then
22:         detach(v, 𝒱_new);                        ▷ detach v from 𝒱_new
23:     else                                         ▷ v was attached to the 𝒱̃_old
24:         for each property p of v do
25:             if p ∈ P_mut && p has RH_diff then
26:                 loadResource(p, RH_diff);
27:         attach(v, 𝒱_new);                        ▷ attach v to 𝒱_new
28: 𝒱̃_new = 𝒱_new;                                 ▷ after updating 𝒱_new, it becomes 𝒱̃_new
29: return 𝒱̃_new;
```

properties might be changed and others might even be added or deleted (i.e., dynamic views). Preserving such changes is critical to the UI consistency. To achieve this, RuntimeDroid needs to update the static view hierarchy and generate another dynamic view hierarchy that is consistent with the one before the runtime change, meanwhile ensuring its compliance with the resource loading. We refer to this process as *dynamic view hierarchy migration.*

To distinguish view properties that might be changed during the user interactions, we introduce *mutable properties.*

Definition 4.6. A view property p is <u>mutable</u> if and only if there exists at least a write operation to the property in at least one callback method of the activity class.

Next, we explain the basic procedure of dynamic view hierarchy migration. For easy references, we use $\mathcal{V}_{\text{old}}$ and $\widetilde{\mathcal{V}}_{\text{old}}$ to represent the old static and dynamic view hierarchies before a runtime change. Correspondingly, we use $\mathcal{V}_{\text{new}}$ and $\widetilde{\mathcal{V}}_{\text{new}}$ to represent the new static and dynamic view hierarchies after the runtime change.

Algorithm 1 illustrates the procedure of dynamic view hierarchy migration. At high level, it has three steps: (i) preparation, (ii) static view migration, and (iii) dynamic view migration. The preparation step identifies the sets of dynamic views, static views, and mutable properties, respectively. In the second step, it migrates the mutable properties of static views from the old static view hierarchy $\mathcal{V}_{\text{old}}$ to the new static view hierarchy $\mathcal{V}_{\text{new}}$. If a mutable property has a resource in RH_{diff} (existence of alternative resource), then a resource loading is performed before the property copying. The third step migrates dynamic views from the old dynamic view hierarchy $\widetilde{\mathcal{V}}_{\text{old}}$ to the new static view hierarchy $\mathcal{V}_{\text{new}}$. When a view was deleted from old static view hierarchy ($\mathcal{V}_{\text{old}}$), it also needs to be detached from the new static view hierarchy ($\mathcal{V}_{\text{new}}$). Similarly, a newly added view needs to be attached to the new static view hierarchy as well. When attaching a dynamic view, it examines if any of its properties are mutable meanwhile has alternative resources (checking RH_{diff}), if so, it first loads the resources before attaching the view.

In practice, the identification of mutable properties (Line 9 in Algorithm 1) can effectively leverage the existence of view IDs. When a view is declared without any assigned ID, then it will not be accessible anywhere from the source code, hence all of its properties become immutable. For event listeners, the dynamic view hierarchy migration treats them as the properties of corresponding views where the listeners are declared. Like other view properties, the event listeners can be mutable, which means they can be attached, detached, or changed by some callbacks.

Figure 6 illustrates the dynamic view hierarchy migration with a simple example, including two dynamically deleted views (`c` and `d`), one added view (`f`), and one mutable property migration (`b.text`). The final result is a new dynamic view hierarchy (the right most).

4.4 Implementations

For easy adoption, we developed two versions of RuntimeDroid: an Android Studio plugin – RuntimeDroid-Plugin and an automatic patching tool – RuntimeDroid-Patch. The former can be used during the app development, while the latter works for compiled Android APK packages.

The implementation of RuntimeDroid follows a modular design with a customized activity class `RActivity`, from which the existing activities in an app can extend. For example, if a developer-defined activity A extends from another activity B

$$A \xrightarrow{\text{extends}} B$$

then RuntimeDroid would refactor it to

$$A \xrightarrow{\text{extends}} \texttt{RActivity} \quad \texttt{RActivity} \xrightarrow{\text{extends}} B$$

Here, some common cases of B include built-in activities, like `Activity`, `AppCompatActivity`, and `FragmentActivity`.

Inside `RActivity`, we implement HotR with the dynamic view hierarchy migration technique mainly by overriding the callback `onConfigurationChanged()`.

This design has two major benefits. First, by extending from the class `RActivity`, all the existing implementation of callback `onConfigurationChanged()` can be preserved by the compiler as the method of a subclass. Second, the class extension provides a modular design which clearly isolates the newly added runtime change handling from the existing implementation of an activity.

Besides introducing the `RActivity` class, RuntimeDroid also parses the manifest file `AndroidManifest.xml` to insert the flag for each type of runtime change into `configChanges` to suspend the activity restarting for all the corresponding runtime changes.

Next, we briefly explain the two implementations.

RuntimeDroid-Plugin. The plugin is implemented on Android Studio 3.0. Developers can use the plugin to refactor a selected activity. The refactoring process automatically insert the `RActivity` into the inheritance hierarchy of the selected activity and injects all the runtime change flags. One challenge for this implementation is that the set of resources may be changed after the refactoring. To address this, the plugin leverages the reflection of `R` class to postpone the identification of available resources to runtime.

RuntimeDroid-Patch. In some situations, one may want to avoid any modifications to the source code or to apply RuntimeDroid

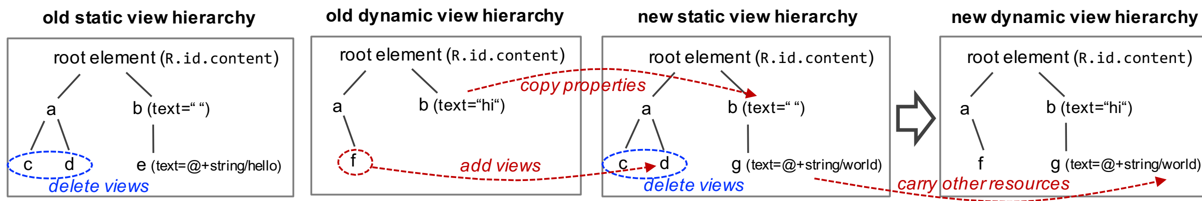

Figure 6: Illustration of Dynamic View Hierarchy Migration.

without the app source code. For such purpose, we implemented RuntimeDroid also as a patching tool. The tool directly takes a compiled Android APK file and injects the customized activity class along with other necessary code into the APK file. More specifically, we leverage Soot [15] and APKtool [17] for reverse engineering and recompilation, and zipalign [18] and Jarsigner [14] to align and sign the processed APK file. The key step – code refactoring – was implemented by ourselves.

5 EVALUATION

This section evaluates RuntimeDroid on its applicability, issue fixing effectiveness, and its impacts in terms of time and space.

5.1 Methodology

The evaluation is performed using *Corpus-S*, which consists of 72 projects, 507 activity instances and 1.5M lines of code. 36/72 apps (i.e., 50%) are also hosted on Google Play Store[22], including some highly popular ones (see Section 3.2 for more details). For each project, we applied RuntimeDroid to each activity that is registered in the `AndroidManifest.xml` file.

In the evaluation, we tested both RuntimeDroid-Plugin and RuntimeDroid-Patch. In order to test RuntimeDroid-Patch, we manually compiled each Android app project in *Corpus-S* with Android Studio 3.0 and generated the APK package. To evaluate RuntimeDroid-Plugin, we loaded each project into an Android Studio IDE with RuntimeDroid-Plugin installed. In order to verify the correctness, we manually checked all the processed activities and examined the app behaviors by deploying the app on a real device – a Nexus 5x smartphone with Android 8.0 installed. The platform for measuring the performance of RuntimeDroid is a Macbook Pro laptop with 2.0 GHz Core i5 processor and 8 GB RAM.

5.2 Applicability

Table 6 summarizes the results of applying RuntimeDroid to the apps in *Corpus-S*. In total, RuntimeDroid was applied to all the 507 activities from 72 projects. Among them, there are 503 activities successfully refactored by RuntimeDroid-Plugin. Only 4 activities (i.e., less than 1%) failed. The reason for that is the 4 activities are from some third-party libraries, in which case the source code of them are not available to RuntimeDroid-Plugin. In comparison, when processing the compiled APK packages with RuntimeDroid-Patch, all the 507 activities are successfully patched. The reason is that RuntimeDroid-Patch does not require accesses to the source code. In fact, the APK package already contains all the compiled code, including the ones of third-party libraries.

Despite the success of processing all the activities, there are a few special cases worth mentioning here. For example, `ListActivity` or `PreferenceActivity`, which do not expose `setContentView()` to developers. In this case, there will be no layout resources or static view hierarchies available. But dynamic views may still be used, which can be detected by HotR (no layout ID found) and handled by the dynamic view hierarchy migration. Another special activity is the `NativeActivity`, which is used mainly for the development of graphics-intensive game apps. Due to the high-performance requirements, these apps often implement their own UI components and event handling mechanisms in C/C++ language, which are not part of Android framework. Though RuntimeDroid can disable the `NativeActivity` from restarting, it will not be able to load resources automatically for any C/C++ defined UI components. In fact, even within the Android framework, developers may define new UI components (views). However, these customized views can still be handled by RuntimeDroid, as long as the property-resource mapping (M_{PR}) for these views are supplemented.

In addition, Table 6 also reports the number of static views in each app V and the number of string resources str. The numbers, to certain extent, reflect the size the view hierarchy and the amount of available resources for some common type of resource. As shown later, these factors may affect the runtime cost of RuntimeDroid. The last column of Table 6 indicates that a large ratio of activities (86%) require resource reference localization, due to global-level declarations of certain resources (see Section 4.2).

5.3 Issue Fixing

We manually examined each reported runtime change issue for each app after applying RuntimeDroid. Both implementations RuntimeDroid-Plugin and RuntimeDroid-Patch are able to fix all the 197 runtime change issues, thanks to the adoption of restarting-free runtime change handling. As discussed earlier (see Section 3.2), activity restarting is the common contributor to the triggering of a variety of runtime change issues. In addition, we did not observe any new issues introduced by the RuntimeDroid, thanks to the HotR and dynamic view hierarchy migration which together preserve UI-resource consistency and the activity state.

Though RuntimeDroid provides a complete coverage of issue fixing for *Corpus-S*, it may not fix all runtime change issues. This is because *not all runtime change issues are caused by activity restarting*. For example, `Firefox` browser displays context menus when long clicking an website icon. However, after a rotation, the context menu gets mispositioned in the screen. This is because after the rotation, both the screen size and the position of icons are changed, while the position setting of the menu is not updated accordingly. Issues like this would still appear even the activity is not restarted.

Note that, alternatively, developers may opt to use data saving and restoring mechanisms (H1.1) or object retaining techniques (H1.2) (see Section 2.2) to fix the issues. However, these solutions

Table 6: Results of applying RuntimeDroid to the activities in Android projects.
A: number of activities, A_s: number of activities successfully processed, str: number of string resources, V: number of static views, I: number of issues, I_f: number of fixed issues, A_l: number of activities requiring resource reference localization

#	App Project	A	A_s	str	V	I	I_f	A_l	#	App Project	A	A_s	str	V	I	I_f	A_l
1	0xbb/otp-authenticator	2	1	14	7	2	2	1	37	jufickel/rdt	1	1	10	25	1	1	1
2	Amabyte/vtu-cs-lab-manual	5	5	3	26	1	1	3	38	julian-klode/dns66	3	3	27	21	1	1	3
3	AntennaPod/AntennaPod	19	19	115	109	5	5	14	39	knirirr/BeeCount	8	8	85	53	2	2	8
4	arnowelzel/periodical	5	5	61	141	2	2	4	40	kraigs-android/kraigsandroid	2	2	15	17	5	5	2
5	artemnikitin/tts-test-app	1	1	4	9	1	1	1	41	liato/android-bankdroid	12	12	69	97	3	3	9
6	awaken/sanity	28	28	147	8	9	9	18	42	LonamiWebs/Stringlate	10	10	117	73	5	5	9
7	balau/fakedawn	3	3	1	39	2	2	3	43	mikifus/padland	10	10	59	33	3	3	5
8	basil2style/getid	2	2	0	34	1	1	2	44	nathan-osman/chronosnap	2	2	16	16	1	1	2
9	benjaminaigner/aiproute	3	3	28	20	3	3	2	45	nbenm/ImapNote2	4	4	0	37	3	3	3
10	blanyal/Remindly	4	4	23	83	2	2	4	46	netmackan/ATimeTracker	5	5	70	21	15	15	4
11	blaztriglav/did-i	2	2	6	9	3	3	2	47	ojacquemart/vlilleChecker	5	4	9	41	1	1	1
12	cbeyls/fosdem-companion-android	8	8	21	24	5	5	7	48	olejon/mdapp	42	42	392	284	2	2	39
13	charbgr/Anagram-Solver	1	1	0	1	2	2	1	49	PaperAirplane.../GigaGet	4	4	13	44	1	1	4
14	charlieCollins/and-bookworm	10	10	113	81	4	4	9	50	peoxnen/GitHubPresenter	1	1	1	3	2	2	1
15	conchyliculture/wikipoff	8	8	58	79	2	2	7	51	phikal/ReGeX	4	4	33	54	3	3	4
16	DF1E/SimpleExplorer	4	4	6	14	1	1	3	52	phora/AeonDroid	5	5	0	44	4	4	2
17	enricocid/Color-picker-library	2	2	4	8	1	1	2	53	phora/AndroPTPB	6	6	0	43	1	1	4
18	erickok/transdroid-search	3	3	0	0	1	1	2	54	pilot51/voicenotify	2	2	41	5	1	1	1
19	EvanRespaut/Equate	2	2	18	3	2	2	2	55	quaap/Primary	11	11	52	67	4	4	10
20	farmerbb/Taskbar	24	24	65	22	3	3	9	56	RomanGolovanov/ametro	6	6	22	34	1	1	6
21	fr3ts0n/StageFever	2	2	1	3	1	1	2	57	rubenwardy/mtmods4android	7	7	54	51	1	1	6
22	gateship-one/malp	5	5	34	51	5	5	4	58	scoute-dich/PDFCreator	8	8	32	22	4	4	7
23	gateship-one/odyssey	4	4	67	25	4	4	3	59	scoute-dich/Sieben	32	32	677	112	2	2	30
24	gianluca-nitti/android-expr-eval	2	2	14	14	1	1	2	60	scoute-dich/Weather	8	8	108	33	3	3	8
25	google/google-authenticator-android	12	12	85	50	3	3	11	61	SecUSo/privacy-friendly-ruler	6	6	22	47	2	2	4
26	grmpl/StepandHeightcounter	2	2	20	5	2	2	2	62	shkcodes/Lyrically	2	2	11	21	2	2	2
27	grzegorznittner/chanu	22	22	114	105	2	2	9	63	SteamGifts/SteamGifts	11	10	20	33	3	3	7
28	hoihei/Silectric	5	5	9	38	2	2	5	64	tarunisrani/InstaHack	2	2	0	13	1	1	2
29	HoraApps/LeafPic	9	9	206	188	2	2	8	65	TeamNewPipe/NewPipe	13	13	39	74	6	6	7
30	HugoGresse/Anecdote	1	1	7	3	1	1	1	66	TobiasBielefeld/Simple-Solitaire	6	6	27	48	1	1	5
31	icasdri/Mather	2	2	5	5	1	1	2	67	ukanth/afwall	14	14	248	101	4	4	13
32	iSoron/uhabits	8	7	6	5	6	6	1	68	vIiRuS/Omnomagon	4	4	32	71	1	1	2
33	JamesFrost/SimpleDo	6	6	26	70	6	6	6	69	VREM.../WiFiAnalyzer	3	3	20	36	2	2	3
34	jiro-aqua/aGrep	6	6	31	25	3	3	5	70	wentam/DefCol	5	5	0	18	1	1	5
35	jparkie/Aizoban	4	4	26	21	1	1	4	71	xargsgrep/PortKnocker	5	5	24	15	4	4	4
36	jpriebe/hotdeath	3	3	5	5	5	5	3	72	zxing/zxing	9	9	41	50	4	4	6

often require significant efforts to manually refactor the app. For example, `ViewModel` and `LiveData` require a redesign of the app to separate the data from the activities. Moreover, they only help address a subset of runtime change issues that are caused by the unsaved data. There are also many problems due to other reasons, which can still be triggered by activity restarting (e.g., menu closing, dialog disappearing, GUI distorting, and asynchronous call caused app crashes). Since the `ViewModel` and `LiveData` do not prevent the activity from restarting, these issues will still occur.

5.4 Handling Efficiency Improvement

Besides fixing runtime change issues, a more general benefit of applying RuntimeDroid is the improvement of runtime change handling efficiency. Due to the invocation of lifecycle callbacks, the conventional restarting-based runtime change handling is often unnecessarily inefficient, not to mention the potential presence of blocking operations in some of the lifecycle callbacks.

The first two columns of Table 7 show the runtime costs for handling a runtime change before and after applying RuntimeDroid. The data clearly shows that the runtime cost is dramatically reduced, 9.5X on average. This is mainly due to the elimination of activity restarting. In fact, for some apps with significant runtime change delays, such as `weather&clock` [9](5-sec delay), `weather` (#60, 3-sec delay), and `GitHubPresenter` (#50, 1-sec delay), the delays would also be dropped to around 20 ms.

In addition, we compared the runtime memory consumption for apps with and without applying RuntimeDroid, with the help of Android Studio Memory Profiler [29]. The measurement injects a series of runtime changes to the apps and collects the memory footprints over a session of 10 minutes, on the tested Nexus 5x smartphone. The results show no observable differences.

Table 7: Handling efficiency and time cost

	Runtime (ms)		Plugin (ms)		
	before	after	1st	2nd	patch (ms)
mdapp	364	57	2291	360	161,598
Remindly	109	21	936	196	43,215
AlarmKlock	117	18	1376	168	12,867
Weather	157	22	1676	543	51,822
PDF Creator	360	10	852	148	94,866
Sieben	126	16	951	155	53,149
AndroPTPB	215	19	627	433	26,708
vlilleChecker	240	21	876	167	56,563
geomean	190	20	1,104	239	49,400

5.5 Time and Space Costs

Time Costs. Table 7 reports different kinds of time costs related to RuntimeDroid. The two columns under "Plugin" report the time spent for refactoring the first and second activities using the RuntimeDroid-Plugin. On average, the time costs are 1,104 ms and 239 ms. The reason that the first activity takes longer time is because it inserts the `RActivity` class and related utility classes for the first time. Note that the utility classes can be shared among all activities of an app. The "patch" column reports the time costs for applying a patch to the whole app APK. This takes from 12 seconds to 2 minutes. The dominate time in applying the patching is the reverse engineering part performed by the Soot.

Table 8: Space cost of RuntimeDroid

	Plugin (SLoC)		Patch (bytes)	
	before	after	before	after
mdapp	26,342	28,419	8,575,378	9,129,420
Remindly	6,966	7,820	1,317,186	1,530,807
AlarmKlock	2,838	3,610	113,037	141,893
Weather	10,949	12,208	3,850,671	4,058,323
PDF Creator	19,624	20,895	10,660,503	10,856,795
Sieben	20,518	22,123	3,945,791	4,203,960
AndroPTPB	3,405	5,127	564,722	596,647
vlilleChecker	12,083	12,843	2,323,633	2,616,449
geomean	9,929	11,463	2,014,635	2,212,115

Space Costs. Table 8 reports the space costs for eight apps from *Corpus-S*. The second and third columns report the source lines of code (SLoC) before and after applying RuntimeDroid-Plugin, respectively. They do not include any library code or non-Java code. On average, the SLoC increases by about 15%. In general, the cost ratio decreases as the app size increases. This is because if multiple activities share the same parent activity, only one copy of `RActivity` is inserted. This amortizes the space cost as the more activities with the same parent activity added.

The last two columns show the sizes of APK files before and after applying RuntimeDroid-Patch. On average, the SLoC increases by about increases by about 10%. For fair comparisons, we recompiled the original apps with our compilation tool chain (see Section 4.4).

6 RELATED WORK

This section summarizes and discusses existing research mostly related to this work, including finding bugs in mobile apps, mobile app refactoring, and API usage study.

6.1 Finding Bugs for Mobile Apps

There exist a large body of work in detecting bugs for mobile apps. AppDoctor [43] injects a sequence of events into an app execution. One of these events is *rotate*, a type of runtime changes. Though mentioning the potential issues during restarting, this work does not offer a systematic solution. Zaeem and others [59] present a mobile app testing tool by deriving test cases from GUI models and interactions. The tool compares the GUI states before and after the interactions, including screen rotation, pausing and resuming, killing and restarting and back key event. Their reported issues include the ones triggered by screen rotations. Adamsen and others [32] inject *neutral* event sequences, such as pause-resume, pause-stop-restart and pause-stop-destroy-create to test apps.

Shan and others [57] propose static and dynamic analysis to discover the Kill and Restart (KR) errors for smartphone apps. This work focuses on discovering and verifying KR errors. In comparison, our work focuses on runtime changes that could trigger KR errors. Also, our work offers a general fixing solution to these issues.

Amalfitano and others [33] study the orientation changes and classes of issues due to orientation changes. They use record & replay technique to match the GUIs after a double-orientation event. They identify several classes of GUI state lost issues, such as Dialog, menu, and view state loss. These findings overlap with some of findings of our work, as orientation is a type of runtime change. Similar to the prior work, it does not provide fixing solutions.

Existing work on app analysis, verification and refactoring mainly focus on other types of issues, including detecting race condition and energy bugs using dynamic analysis [41, 42, 49], uncovering bugs with network and location data [46], detecting performance bugs [34, 36, 47, 48, 50] and memory leaks [58].

6.2 API Usage and Mobile App Refactoring

There are also empirical studies on programming languages and libraries usages [37–40, 44, 52]. Buse and others [37] introduce an automatic technique for mining and synthesizing documents for program interfaces. Kavaler and others [45] investigate Android APIs questions on Stackoverflow [6]. Unlike prior work, this work studies the APIs and practices of runtime change handling.

Bavota and other [35] study the refactoring activities and their impacts, including the potential of refactoring-induced faults. In addition to functional refactoring, there is a trend in refactoring for non-functional qualities, like refactoring built-in locks with more flexible alternatives [56], refactoring global state with thread-local state [55], refactoring the concurrent programming constructs [51], and refactoring for energy efficiency [54]. Unlike the objectives of prior work, this work aims for a refactoring-based solution for addressing issues in runtime change mishandling.

7 CONCLUSION

Unlike traditional desktop applications, mobile apps experience more frequent runtime changes. When handled inappropriately, such simple runtime changes may cause critical issues. In this work, we present, to our best knowledge, the first formative study on the runtime change handling for Android apps. The study not only reveals the current landscape of runtime change handling, but also identifies a common cause for a variety of runtime change issues – *activity restarting*. With this insight, it introduces a restarting-free runtime change handling solution, named RuntimeDroid, which can load resources without restarting the activity. It achieves with this with an *online resource loading module* called HotR. More critically, it can preserve prior UI changes with a novel *dynamic view hierarchy migration* technique.

For easy adoption, this work provides two implementations, RuntimeDroid-Plugin and RuntimeDroid-Patch, to cover both in-development and post-development uses for Android apps. The evaluation shows that RuntimeDroid can successfully refactor 503/507 activities and fix 197/197 real-world runtime change issues, meanwhile reducing the handling delays by 9.5X on average.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their constructive comments. We also thank our shepherd Dr. Landon Cox for his guidance and valuable suggestions during the preparation of the final version.

REFERENCES

[1] 2016. Alarm Klock. https://play.google.com/store/apps/details?id=com.angrydoughnuts.android.alarmclock. (2016). Accessed: 2016-04-22.
[2] 2016. Github. https://github.com/. (2016). Accessed: 2016-04-22.
[3] 2016. ImapNote 2. https://github.com/nbenm/ImapNote2. (2016). Accessed: 2016-04-22.
[4] 2016. Multi-Window Support. https://developer.android.com/guide/topics/ui/multi-window.html. (2016). Accessed: 2017-11-12.
[5] 2016. Resource Types. https://developer.android.com/guide/topics/resources/available-resources.html. (2016). Accessed: 2017-11-12.
[6] 2016. StackOverflow. https://stackoverflow.com/. (2016). Accessed: 2016-04-22.

[7] 2016. Using a Hardware Keyboard With an Android Device. https://www.nytimes.com/2016/03/29/technology/personaltech/using-a-hardware-keyboard-with-an-android-device.html. (2016). Accessed: 2017-11-12.

[8] 2016. Vlille Checker. https://play.google.com/store/apps/details?id=com.vlille.checker. (2016). Accessed: 2016-04-22.

[9] 2016. Weather & Clock Widget for Android. https://play.google.com/store/apps/details?id=com.devexpert.weather. (2016). Accessed: 2016-04-22.

[10] 2017. Android Bundle. https://developer.android.com/reference/android/os/Bundle.html. (2017). Accessed: 2017-12-01.

[11] 2017. Android SharedPreferences. https://developer.android.com/reference/android/content/SharedPreferences.html. (2017). Accessed: 2017-12-01.

[12] 2017. Android SharedPreferences Editor. https://developer.android.com/reference/android/content/SharedPreferences.Editor.html. (2017). Accessed: 2017-12-01.

[13] 2017. Android SQLite Database. https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html. (2017). Accessed: 2017-12-01.

[14] 2017. jarsigner - JAR Signing and Verification Tool. http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html. (2017). Accessed: 2016-04-22.

[15] 2017. Soot: a Java Optimization Framework. https://www.sable.mcgill.ca/soot/. (2017). Accessed: 2017-12-01.

[16] 2017. Supporting Different Languages and Cultures. https://developer.android.com/training/basics/supporting-devices/languages.html. (2017). Accessed: 2017-11-12.

[17] 2017. A tool for reverse engineering Android apk files. http://ibotpeaches.github.io/Apktool/. (2017). Accessed: 2016-04-22.

[18] 2017. zipalign: an archive alignment tool. https://developer.android.com/studio/command-line/zipalign.html. (2017). Accessed: 2017-12-01.

[19] 2018. Android Architecture Components. https://developer.android.com/topic/libraries/architecture/. (2018). Accessed: 2018-04-22.

[20] 2018. Barcode Scanner. https://play.google.com/store/apps/details?id=com.google.zxing.client.android. (2018). Accessed: 2018-04-22.

[21] 2018. The Google I/O 2017 Android App. https://github.com/google/iosched. (2018). Accessed: 2018-04-22.

[22] 2018. Google Play Store. https://play.google.com/store?hl=en. (2018). Accessed: 2018-04-22.

[23] 2018. K-9 Mail âĂŞ Advanced Email for Android. https://github.com/k9mail/k-9/. (2018). Accessed: 2018-04-22.

[24] 2018. Loop - Habit Tracker. https://play.google.com/store/apps/details?id=org.isoron.uhabits. (2018). Accessed: 2018-04-22.

[25] 2018. Material design file manager for Android. https://github.com/TeamAmaze/AmazeFileManager. (2018). Accessed: 2018-04-22.

[26] 2018. Material Design Music Player. https://github.com/naman14/Timber. (2018). Accessed: 2018-04-22.

[27] 2018. Smartphone market share. http://www.idc.com/promo/smartphone-market-share/. (2018). Accessed: 2018-04-11.

[28] 2018. Telegram for Android source. https://github.com/DrKLO/Telegram. (2018). Accessed: 2018-04-22.

[29] 2018. View the Java Heap and Memory Allocations with Memory Profiler. https://developer.android.com/studio/profile/memory-profiler. (2018). Accessed: 2018-04-29.

[30] 2018. WiFiAnalyzer. https://play.google.com/store/apps/details?id=com.vrem.wifianalyzer. (2018). Accessed: 2018-04-22.

[31] 2018. WordPress for Android. https://github.com/wordpress-mobile/WordPress-Android. (2018). Accessed: 2018-04-22.

[32] 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*. ACM, 83–93.

[33] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Softw. Test., Verif. Reliab.* 28, 1 (2018).

[34] Niaz Arijo, Reiko Heckel, Mirco Tribastone, and Stephen Gilmore. 2011. Modular performance modelling for mobile applications. In *ACM SIGSOFT Software Engineering Notes*, Vol. 36. ACM, 329–334.

[35] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. IEEE, 104–113.

[36] Luca Berardinelli, Vittorio Cortellessa, and Antinisca Di Marco. 2010. Performance modeling and analysis of context-aware mobile software systems. *Fundamental Approaches to Software Engineering* (2010), 353–367.

[37] Raymond PL Buse and Westley Weimer. 2012. Synthesizing API usage examples. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 782–792.

[38] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2013. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering* 18, 6 (2013), 1156–1194.

[39] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. 2014. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 779–790.

[40] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 2010. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 11.

[41] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. 2014. Race detection for event-driven mobile applications. *ACM SIGPLAN Notices* 49, 6 (2014), 326–336.

[42] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 77–83.

[43] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 18.

[44] Siim Karus and Harald Gall. 2011. A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 13–22.

[45] David Kavaler, Daryl Posnett, Clint Gibler, Hao Chen, Premkumar T Devanbu, and Vladimir Filkov. 2013. Using and Asking: APIs Used in the Android Market and Asked about in StackOverflow.. In *SocInfo*. Springer, 405–418.

[46] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. 2014. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*. ACM, 519–530.

[47] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 445–456.

[48] 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*. ACM, 1013–1024.

[49] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for android applications. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 316–325.

[50] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press, 29–35.

[51] Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. 2014. A study and toolkit for asynchronous programming in C. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1117–1127.

[52] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and use of Java generics. *Empirical Software Engineering* 18, 6 (2013), 1047–1089.

[53] Alireza Sahami Shirazi, Niels Henze, Tilman Dingler, Kai Kunze, and Albrecht Schmidt. 2013. Upright or sideways?: analysis of smartphone postures in the wild. In *Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services*. ACM, 362–371.

[54] Cagri Sahin, Lori Pollock, and James Clause. 2014. How do code refactorings affect energy usage?. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 36.

[55] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. 2010. Correct refactoring of concurrent java code. *ECOOP 2010–Object-Oriented Programming* (2010), 225–249.

[56] Max Schafer, Manu Sridharan, Julian Dolby, and Frank Tip. 2011. Refactoring Java programs for flexible locking. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 71–80.

[57] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 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*. ACM, 864–880.

[58] Dacong Yan, Shengqian Yang, and Atanas Rountev. 2013. Systematic testing for resource leaks in Android applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 411–420.

[59] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. 2014. Automated generation of oracles for testing user-interaction features of mobile apps. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 183–192.