Umar Farooq and Zhijia Zhao *Department of Computer Science and Engineering, University of California, Riverside*

Editors: Nic Lane and Xia Zhou

RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps

Illustration, istockphoto.com

Excerpted from "RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps" from MobiSys 2018, *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications and Services* with permission. https://dl.acm.org/citation.cfm?id=3210327 © ACM 2018

Portable devices, like smartphones and tablets, are often subject to higher frequency of configuration changes, such as screen orientation changes, screen resizing, keyboard attachments, and language switching. Since the changes can happen at runtime while users interact with the devices, they are referred to as *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 five minutes accumulatively over sessions of the same app [1]. For multilingual or tablet users, changing the language setting or attaching an external keyboard is often desired [2,3]. As newer versions of Android systems with multiwindow supports are adopted, it is projected that runtime changes will happen more frequently. Each time a user drags the boundary between two split windows, a runtime change would be triggered [4].

When handled improperly, such simple configuration changes can cause serious runtime issues, from user data loss to app crashes. In this work, we present, to our best knowledge, the first formative study on runtime change handling of real-world 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* (an activity in Android represents an interactive screen). Based on the findings, we design and implement a restarting-free runtime change handling solution – *RuntimeDroid*, which automatically prevents the activities from restarting, but ensuring proper resource updating and user data preservation. By avoiding activity restarting, RuntimeDroid successfully fixed a set of 197 reported runtime change issues, meanwhile reducing the runtime change handling delays by 9.5X on average.

RUNTIME CHANGE AND ITS HANDLING

Table 1 lists the runtime changes defined by the Android API (Level 25). For example, a device rotation will trigger both screen orientation and screen size changes (since Android 3.2) and window resizing in multi-window mode will trigger screen size changes (since Android 7). Besides screen-related changes, there are also runtime changes for cellular networks, keyboard availability, language, font size, and layout direction.

During a runtime change, an app needs to load resources for the new configuration. For example, when the screen is rotated from `portrait` to `landscape`, a different layout designed for the `landscape` needs to be loaded (if available), which may carry UI elements with adjusted dimensions. In general, developers can provide a wide spectrum of resources, from strings and colors to images and layouts, for different device configurations. All these resources are grouped and placed in the 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	Layout Direction

To effectively handle various runtime changes and load needed resources accordingly, Android offers two basic strategies: *restarting-based handling* (default) and *customized handling*. By default, Android would first destroy the current activity, then start 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 1). In many cases, an activity may carry the user interaction state, such as a selected article in a news app or the player score of a game app. When the old activity gets destroyed, its state is also wiped. To avoid losing the user interaction state, developers need to preserve state-critical data during the activity restarting. The preservation can be achieved by either saving/restoring the activity state with APIs `onSave/RestoreInstanceState()` or retaining the data objects with some special Android constructs (`Fragments` and `LiveData`).

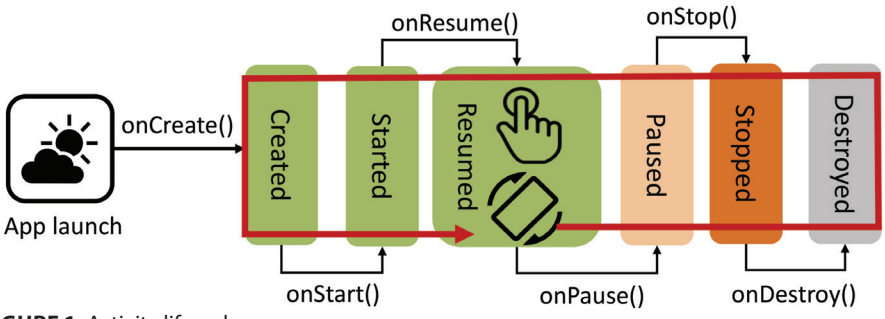


FIGURE 1. Activity lifecycle.

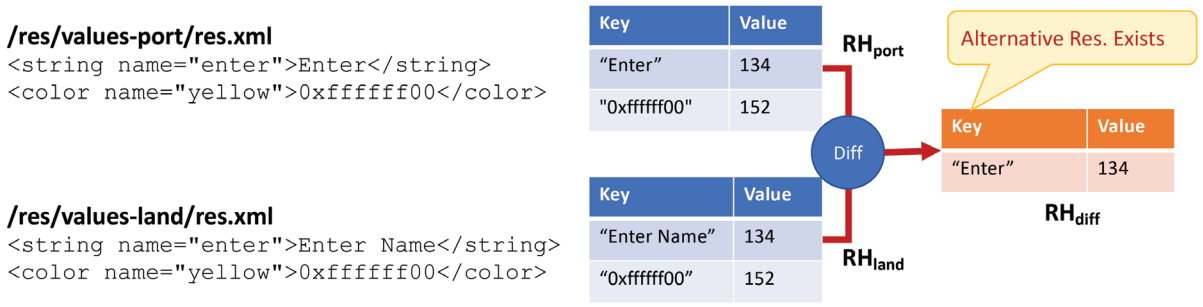


FIGURE 2. Resource construction.

Instead of letting the activity restart, developers may choose to directly program the runtime change handling (customized handling). To do so, developers need to set the runtime change flag `android:configChanges` for self-handling changes in the app configuration file (i.e., `AndroidManifest.xml`). Once flagged, a runtime change will no longer trigger any activity restarting. Instead, it will invoke `onConfigurationChanged ()` callback. By overriding this callback, developers can manually load alternative resources for the new configuration. However, manual resource loading is complex to implement, given the diversity of resource types (16 types in API 27) and their broad uses.

LANDSCAPE OF RUNTIME CHANGE HANDLING

To find the common practices of runtime change handling in real-world Android apps, we collected a set of 3,567 popular Android apps from Github (based on the number of stars) with a total of 16,160 activities, referred to as *Corpus-L*. To facilitate the study, we developed an automatic code analysis tool – *RuntimeAnalyzer*. For each app in the corpus, RuntimeAnalyzer parses its source code and collects the runtime change handling strategy for each registered activity.

The study results reveal that the most common runtime changes concerned by developers are screen orientation change (32.0%), keyboard availability change (26.8%), and screen size change (22.9%). As to the handling strategy, 92.4% of the 16,160 activities choose restarting-based handling, which covers 92.3% of the 3,567 apps. This is mainly due to its lower barriers to program than the customized handling, which requires the understanding of resource loading mechanisms. Among the activities with restarting-based handling, only 13.9% leverage the callback `saveInstanceState ()` to preserve the data and 15.4% adopt object retaining. That means a large portion of the activities (68.3%) provide no mechanisms for data preserving at all. Among the activities that choose the customized handling, only about one third (31.7%) actually override the callback `onConfigurationChanged ()`. The results indicate that a large ratio of Android apps might not be well prepared for the activity restarting, thus making them vulnerable to various runtime change issues, as we show next. In addition, the study shows that a small ratio of activities (15.5%) are set with a fixed orientation. However, this setting can only avoid orientation-related activity restarting, at the cost of restricted user experiences.

RISE OF RUNTIME CHANGE ISSUES

Our preliminary examination of 765 repositories from Github shows that 342 of them (44.7%) had at least one issue due to runtime change mishandling. To characterize the issues and identify their causes, we performed a focused study on a set of 197 runtime change issues from 72 Android apps (referred to as *Corpus-S*). Half of the apps are also hosted on Google Play Store [5], including a few highly popular ones, such as Barcode Scanner [6] (100M+ installs). All the issues are reported in the Github issue tracking system. Based on their manifestation, we categorize them into four basic types.

(i) Lost State: This is the most common type of issue. Examples include losing user inputs, scrolling positions, or opened dialogs.

RUNTIMEDROID CAN LOAD RESOURCES WITHOUT RESTARTING THE ACTIVITY

When an activity is destroyed, its associated UI elements are also removed together along with their attributes, like text, selection, and position. For some built-in UI elements, the system will save and restore certain attributes (e.g., text in `EditText`). However, this may not cover all the UI states, not to mention the Non-UI data. Furthermore, the study shows that despite the saving and restoring, the data might be reset during activity restarting (e.g., by an initialization callback `onCreate ()`).

(ii) Malfunctioning UI: In the setting view of Vlille Checker [7], an app for self-service biking, runtime changes result in two layers of GUIs overlapped with each other. In this particular case, when a runtime change occurs, a new activity is started with a new `Fragment` attached. Meanwhile, the old `Fragment` is still retained by the system, overlapped with the new one. In general, the malfunctioning UI issues are often caused by the improper handling of UI elements during the activity restarting.

(iii) App Crash: The most severe types are app crashes. When an app crash happens, a message "Unfortunately your app has stopped" pops up on the screen. They are often triggered by the misuse of asynchronous function calls (e.g., `AsyncTask`) with restarting-based handling. Basically, an

AsyncTask instance was created before a runtime change. When it finishes after the runtime change, it cannot find the objects in the destroyed activity, thus throwing a NullPointer exception.

(iv) Poor Responsiveness: Some mobile apps exhibit slowness during a runtime change, but users tend to not report them as "issues." Essentially, the delay is caused by some blocking operations (e.g., file/network accesses) in the lifecycle callbacks. When the screen is rotated, the activity gets restarted and the screen becomes irresponsive until the blocking operations finish.

On one hand, runtime change issues exhibit a variety of consequences. On the other hand, they often share a common condition – the adoption of the restarting-based handling. So, a natural question raises: "Can we avoid the activity restarting, while still loading resources as needed?"

RUNTIMEDROID

To address the challenges in runtime change handling, we introduce a restarting-free solution – RuntimeDroid. At a high level, it consists of an online resource loading module – *HotR* and a *dynamic view hierarchy migration* technique.

As mentioned earlier, to prevent activities from restarting during runtime changes, developers can choose the customized handling. However, this requires developers to manually load resources for the new configurations, which is challenging for many Android developers due to the complexities in the types of resource and the dynamic nature of UI elements.

Next, we present an automatic online resource loading module – *HotR*, which is able to load resources for the new configuration while the current activity remains live. Moreover, it does not depend on the app logic.

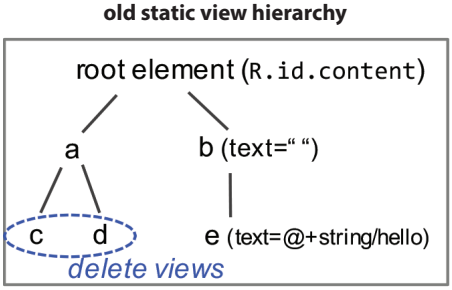
First, for each configuration, HotR constructs a *resource HashMap* with an entry for each declared resource. The key of this HashMap is the (serialized) "content" of a resource and the value is the resource ID. For example, the resources of two configurations in Figure 2 will be compiled into the two resource HashMaps: RH_{port} and RH_{land}.

When a runtime change occurs, HotR calculates the differences between the resource HashMap of the old configuration RH_{old} and the new one RH_{new}, *that is,* $RH_{diff} = RH_{new} - RH_{old}$. If RH_{diff} is non-empty, then HotR would consider the existence of alternative resources, hence triggering the resource loading.

To perform resource loading, HotR distinguishes two cases: resource loading for *static UI elements* and resource loading for *dynamic UI elements*. The static UI elements are pre-defined in the layout XML file. To load their resources, HotR leverages the setContentView () API, which not only loads the new layout and the UI elements declared in the layout, but also the needed resources for their properties. For dynamic UI elements, which include the UI elements added or deleted during the user interaction and the ones with *user-changeable* properties, the resource loading becomes more complicated. We address this complexity with a dynamic view hierarchy migration technique. The high-level idea is illustrated by Figure 3.

The UI elements on the screen form a tree structure, called view *hierarchy*. To systematically update resources for the dynamic UI elements without wiping off the user interaction state, we leverage three versions of view hierarchy: (i) *old static view hierarchy* - the one generated from layout XML file of the old configuration, (ii) *old dynamic view hierarchy* - the actual one before the runtime change, and (iii) *new static view hierarchy* - the one generated from the layout XML file of the new configuration. The basic strategy is to use the new static view hierarchy as the template and update it based on the differences between the two old view hierarchies (i.e., user state). The result is the *new dynamic view hierarchy* - the one that the user is expected to observe after the runtime change (see Figure 3).

In addition, there are two complexities worth mentioning. One is mapping the resources to the properties of UI elements. We address this with a pre-defined mapping called *property-resource mapping*, which is constructed based on the programming conventions (e.g., the text property is mapped to a string resource). The other complexity is that there might be references to the resources in the app logic code (written in Java). For example, the following statement accesses a string resource from a Java method, String hello = getString(R.string.hello); When



the alternative resources are loaded, we need to make sure that the corresponding references point to the newly loaded resources, instead of the old ones. We discuss this complexity and the solution in [8].

IMPLEMENTATION AND EVALUATION

For easy adoption, we developed two versions of RuntimeDroid: An Android Studio refactoring plugin – RuntimeDroid-Plugin and a binary patching tool – RuntimeDroid-Patch. The former can be used during the app development, while the latter works for compiled Android APK packages, enabled by a set of reverse engineering techniques. Both implementations follow a modular design with a customized activity class RActivity, from which the existing activities in an app can extend. For example, a developer-defined activity A extending from another activity B will be refactored as Figure 4. 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 ().

We evaluated RuntimeDroid with the *Corpus-S*, which consists of 72 Android apps with 507 activities and 197 runtime change issues. The evaluation results show that RuntimeDroid-Plugin successfully refactored 503 activities, with 4 activities failed due to the lack of source code (declared in third-party libraries). In comparison, RuntimeDroid-Patch refactored all the 507 activities thanks to its ability of reverse engineering. Note that, despite the success of processing

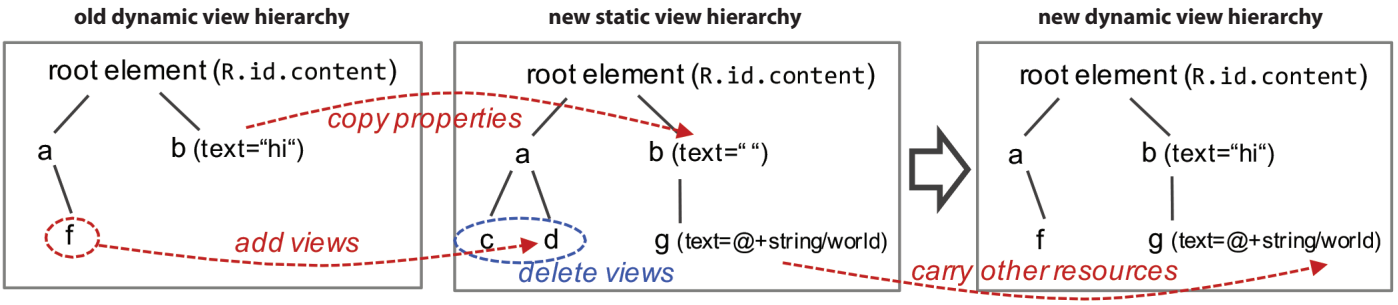


FIGURE 3. Illustration of dynamic view hierarchy migration.

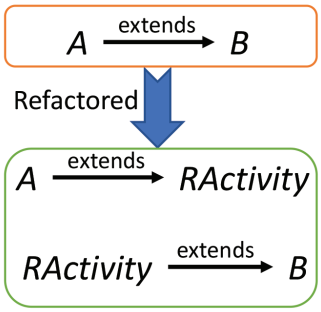


FIGURE 4. Code refactoring by RuntimeDroid.

all the activities in *Corpus-S*, there are a few special cases where RuntimeDroid fails to refactor. These include activities without `setContentView()` API, such as `ListActivity` and `PreferenceActivity`, and activities that are not written in Java (i.e., `NativeActivity`).

More importantly, our evaluation also shows that, by applying RuntimeDroid to the 72 problematic apps, all the 197 reported runtime issues get fixed, thanks to the adoption of restarting-free runtime change handling. Note that, according to our formative study, not all runtime change issues are caused by activity restarting. For example, in Firefox browser, after a screen rotation, its context menu gets mispositioned in the screen. This is because the screen size and the icon position are both changed, while the position of the menu is not updated accordingly. Issues like this would still appear even if the activity is not restarted.

Besides issue fixing, our evaluation also shows the performance benefits of applying RuntimeDroid. By avoiding activity restarting, we observe that the runtime change handling cost is reduced by 9.5X on average. On the other hand, RuntimeDroid may introduce space costs due to the refactoring. However, based on our measurements, the cost is about 15% on average and the ratio decreases as the package size increases. The time for applying RuntimeDroid-Plugin ranges from hundreds of milliseconds to 1 second and the time for applying RuntimeDroid-Patch ranges from 12 seconds to 2 minutes.

CONCLUSION

This work, to our best knowledge, presents the first formative study on the runtime change handling for Android apps. The study reveals the current landscape of runtime change handling and a common cause 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 this with an online resource loading module HotR and 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. Finally, the evaluation confirms the effectiveness and efficiency of RuntimeDroid by refactoring and fixing a set of 197 real-world runtime change issues. ■

Umar Farooq is a PhD candidate in the Computer Science and Engineering Department at University of California, Riverside. His research interests lie broadly in mobile systems and applications, with a focus on designing and applying program analysis and refactoring techniques to address real-world issues in mobile systems and applications. He is a recipient of Best Paper Runner-up Award at MobiSys 2018.

Zhijia Zhao is an assistant professor of Computer Science and Engineering at University of California, Riverside. Prior to joining UCR, he received his PhD from the College of William and Mary in 2015. His research focuses on programming system supports for parallel computing and mobile computing. He is a recipient of an NSF CAREER Award, a Regents Faculty Fellowship, and a Hellman Fellowship.

REFERENCES

[1] 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.

[2] 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 (2017). Accessed: 2017-11-12.

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

[4] Multi-Window Support. https://developer.android.com/guide/topics/ui/multi-window.html (2017). Accessed: 2017-11-12.

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

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

[7] Vlille Checker. https://play.google.com/store/apps/details?id=com.vlille.checker (2018). Accessed: 2018-04-22.

[8] Umar Farooq and Zhijia Zhao. 2018. RuntimeDroid: "Restarting-free runtime change handling for Android apps." In *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services*. ACM, 110–122.