UNIVERSITY OF CALIFORNIA
RIVERSIDE

Runtime, Analysis, and Tools for Reliable Management of Mobile App States

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Umar Farooq

December 2021

The Dissertation of Umar Farooq is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to take this opportunity to thank Prof. Zhijia Zhao who supervised my doctoral studies and research work at the Department of Computer Science and Engineering at UCR. I am grateful to him for his unwavering support and belief in me throughout. I am also thankful to Prof. Manu Sridharan for his unparalleled support and guidance at every step throughout my PhD journey. In addition to his guidance in a professional capacity, I must say that he is a wonderful and compassionate human being, and I believe that my journey of doctoral studies could not have been so successful without his dedicated support.

I am also thankful to my elder brother Dr. Muhammad Abubakar Siddique and Dr. Fuad Jamour for their exceptional brainstorming sessions and assistance in managing an incredibly fruitful research collaboration. Last but not least, I would like to express my gratitude towards Prof. Nael Abu-Ghazaleh, Prof. Rajiv Gupta and Prof. Zhiyun Qian for their unmatched commitment and valuable feedback to strengthen this dissertation and transform it into a valuable contribution to the field of computer science.

I am grateful to all faculty members of the Department of Computer Science and Engineering at UCR, who helped me during classes and navigating through graduate studies would not have been possible without their help. Finally, I would like thank to current and past members RIPLE research group, especially those who are part of the Winston Chung-226 lab. Winston Chung-226 has been a great place to work, collaborate, and have fun at the same time.

To my family for their *unconditional* love and support.

ABSTRACT OF THE DISSERTATION

Runtime, Analysis, and Tools for Reliable Management of Mobile App States

by

Umar Farooq

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2021
Dr. Zhijia Zhao, Chairperson

Unlike traditional computing platforms (such as desktops and servers), mobile platforms provide a volatile running environment for apps where the state of an app may get destroyed and recreated frequently, either by runtime configuration changes (e.g., changing the phone orientation) or by limited resources available on the device (e.g., low memory). When the app state is not saved and restored appropriately, the app restarts may cause a wide range of runtime issues, ranging from loss of user progress and poor responsiveness to malfunctioned UI and app crashes. The goal of this thesis is to address the fundamental issue of state management for mobile apps. To achieve the above goal, this thesis proposes a runtime system and a novel static analysis.

First, it studies a large number of real-world mobile apps, a high percentage of them fail to handle restarts appropriately, posing a great challenge to mobile computing. To address the state issues caused by app restarts, this thesis explores two orthogonal directions: (i) a system-oriented solution and (ii) an application-oriented solution.

In the first direction, it aims to prevent apps from restarting by developing a restart-free runtime system, called RuntimeDroid, which completely avoids the needs of configuration-related app restarts. The purpose of restarts in the existing runtime system is to load resources for the new configuration (e.g., a layout for the portrait mode). RuntimeDroid achieves this by automatically identifying and "hot"-loading the resources while the app is running.

In the other direction, this thesis approaches the state issues with a more direct solution – first, identify the necessary app state, then automatically preserve it at runtime. More specifically, it first uses static analysis to systematically reason about the app source code and the associated resource files to find the critical program variables and properties of GUI elements that are necessary to be preserved. Then, it leverages automatic code generation techniques to insert the state saving and restoring routines to the app code. As a result, a tool (named LiveDroid) can precisely identify and preserve the app state in real-world Android apps, which not only ensures the state correctness during app restarts but also makes the app restarts more responsive.

Besides the above, this thesis also proposes the static user transaction graph (SUTG), which leverages static analysis to encode operations of interests in the form of a graph. SUTG can serve as the basis for a variety of mobile app testing tools, such as latency profiling and state testing in the presence of asynchronous threads. As a result, the automated tools help developers to identify performance bottlenecks and incorrect state handling in their apps.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smartphones are in wide use (there were 3.2 billion smartphone users worldwide in 2019 [150]) and mobile apps have a substantial economic impact (the mobile app market is projected to reach \$407 billion by 2026 [48]). Hence, there is an impetus for ensuring and improving mobile app reliability. Building reliable mobile apps poses additional complications when compared to desktop/server applications, due to the challenges imposed by *rich yet volatile* mobile runtime environments.

Unlike desktop or server applications, mobile apps run in a more challenging environment: devices are resource-limited, and the underlying OS subjects the app to a richer set of disruptive events. Consequently, mobile apps often go through multiple *lifecycles* – being destroyed and recreated – before they are explicitly dismissed. For Android apps as shown in Figure 1.1, when a runtime configuration change occurs, like a phone rotation (portrait $\leftrightarrow$ landscape) or attaching a keyboard, the OS destroys the current screen instance (a.k.a *activity* in Android), including both the GUI elements and (Java) class associated

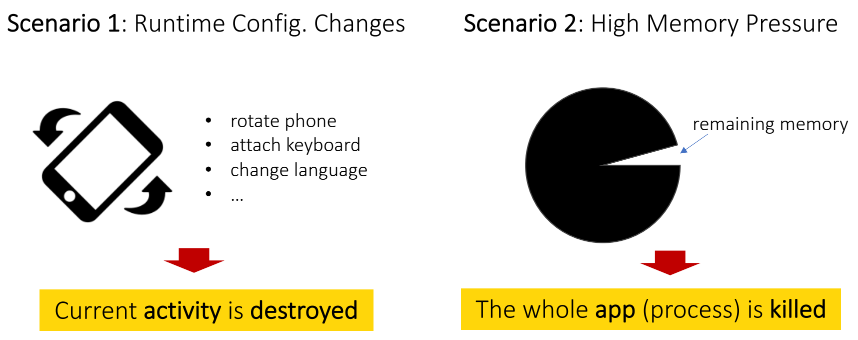

Scenario 1: Runtime Config. Changes
Scenario 2: High Memory Pressure
rotate phone
attach keyboard
change language
...
remaining memory
Current activity is destroyed
The whole app (process) is killed

Figure 1.1: Restarting scenarios for mobile apps at runtime.

with the screen, and then recreates a new screen instance. This process is known as *activity restarting*. The purpose of activity restarting is to automatically reload the activity with resources that match the new configuration (e.g., landscape mode layout after rotation) [95]. Another destructive scenario involves low resources: a running app (especially when sent to the background) can be killed at any time by the OS when memory runs low, then relaunched when the user comes back to the app [65, 74]. This is due to mobile OSes, including both iOS and Android, eschewing swapping (i.e., paging out) [65, 70], to minimize flash memory wear [73]. When an app is killed due to low memory, all its running activities are destroyed.

To avoid losing user progress, or entering into an inconsistent state, certain program variables and properties of the GUI elements must be saved before the activity is destroyed (or app is killed) and restored after the activity gets recreated (or app gets relaunched), as if the activity (or app) remains running in the same lifecycle [96, 71]. Moreover, testing apps states and measuring performance during volatile and asynchronous natures brings additional challenges. This dissertation makes the following contributions to address the challenges to manage app state reliably.

1.1 Dissertation Contributions

Chapter 2. provides a brief background to the relevant programming model, their runtime environment and state management of mobile apps.

Chapter 3. presents, 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 chapter 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 chapter 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.

Chapter 4. proposes a systematic solution, LiveDroid, which precisely identifies the *necessary* part of the app state that needs to be preserved across app lifecycles, and automatically saves and restores it. LiveDroid consists of: (i) a static analyzer that reasons about app source code and resource files to pinpoint the program variables and GUI properties that represent the necessary app state, and (ii) a runtime system that manages the state saving and recovering. We implemented LiveDroid as a plugin in Android Studio and a patching tool for APKs. Our evaluation shows that LiveDroid can be successfully applied to 966 Android apps. A focused study with 36 Android apps shows that LiveDroid identifies app state much more precisely than an existing solution that includes all mutable program variables but ignores GUI properties. As a result, on average, LiveDroid is able to reduce the costs of state saving and restoring by 16.6X (1.7X - 141.1X) and 9.5X (1.1X - 43.8X), respectively. Furthermore, compared with the manual state handling performed by developers, our analysis reveals a set of 46 issues due to incomplete state saving/restoring, all of which can be successfully eliminated by LiveDroid.

Chapter 5. introduces the concept of *static user transaction graph* (SUTG), which statically captures user interface (UI) updates and the effects of the updates in form of a graph. A SUTG can be automatically constructed based on UI updates, the causality relations among asynchronous callbacks, with the help of static analysis. Once, a SUTG is built for a given app, we can utilize it for a variety of applications such as instrumentation for profiling and testing. SUTG works in two modes, developers either can annotate the app events to generate SUTG for specific event, alternatively, our tool would consider all the user action events defined in the app. We provide two applications of SUTG (i) an automatic latency profiling tool, and (ii) an automatic state-management tool for asynchronous operations testing. We applied our tool on 791 Android apps collected from F-Droid and Google Play stores, and evaluations show the effectiveness and applicability of our SUTG. Furthermore, we conducted a focused study on 44 apps to show the effectiveness of SUTG, and its applications, the latency profiler and idling resource manager.

1.2 Dissertation Organization

This dissertation contains published work. Chapter 3 is based on a paper accepted for publication at the International ACM International Conference on Mobile Systems, Applications, and Services `(MobiSys'18)` [91]. Chapter 4 is based on a paper published in the Proceedings of the ACM on Programming Languages `(Issue OOPSLA'20)` [92]. Chapter 5 is based on a to-be-submitted work which defines static user transaction graph and explores its applications.

Chapter 2

Background and Related Work

2.1 App Programming Model

Android apps are primarily written in Java and built around the concept of *activities, services, content providers, and broadcast receivers* referred as *Android components.* 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 2.1. By overriding the lifecycle 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).

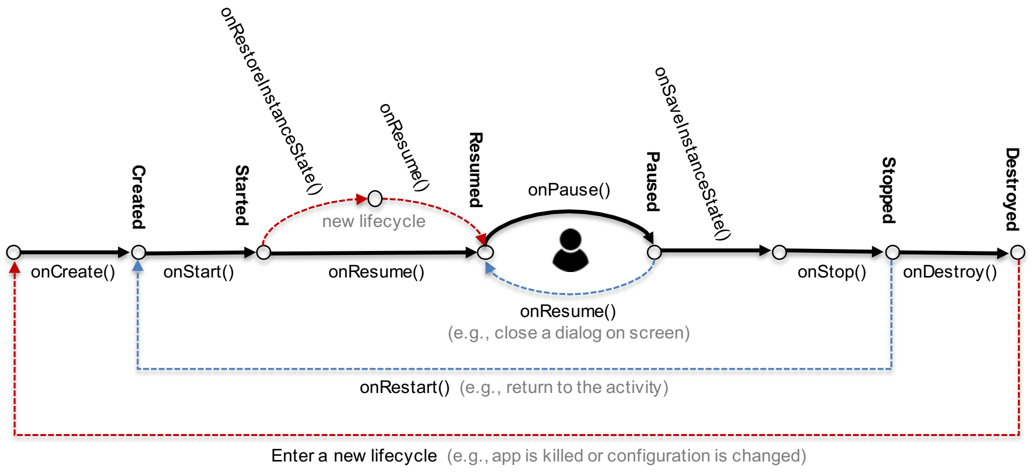


Figure 2.1: Activity Lifecycle.

Service Lifecycle. The services do not expose any interface and a Java class typically implements its logic. Similar to activities, services go through sequence of lifecycle callbacks, including *onCreate(), onStartCommand(), onBind(), onUnbind(),* and *onDestory()* [66]. Android provides two variants of service, bound and started (unbound) services, and a service can be both bound and started as well. Both types have a little lifecycle difference, a bound service invokes *onBind() and onUnbind()* to establish and disconnect a connection, whereas a started service invokes *onStartCommand()* at the start and requires explicit `stopService()` or `stopSelf()` call to stop a started service. The bound services stay alive as long as binding exists, whereas started services might require explicit stop call depending on the flag. Finally, both types invoke `onDestroy()` callback followed by unbind or stop request.

Broadcast receivers and content providers Lifecycle. The content providers and broadcast receivers offer simpler lifecycle than activities and services. Typically, content providers go though `onCreate()` callback to perform lightweight initialization, furthermore apps implement data manipulation callbacks (e.g., `query()` and `insert()`) [54]. The broadcast receivers offer much simple flow, there is only one method `onReceive()` that gets invoked when (Android) system delivers a broadcast [53].

Inter-component control flow. All app components can invoke and communicate using a special type of messages, called *intent*. For example, an activity can start another activity using `startActivity(Inent)`, and start a service by `startService(Intent)`. The invoked component starts within its own lifecycle, and component used to invoke stays within its lifecycle, for instance when an activity A invokes an activity B, the activity A completes the execution of current callback and calls `onPause()` callbacks since user moved to the activity B. Then activity B starts its lifecycle and executes `onCreate()` callback and so on. When user returns to the activity A, lifecycle continues by invoking `onResume()` of activity A and making activity A available for user interaction.

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

Table 2.1: 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

respond to the events. This single-thread model requires developers to limit the workload of UI thread to keep the app responsive.

2.2 Volatile Runtime Environments

Unlike conventional desktop applications, Android apps (and activities) may go through multiple lifecycles (i.e., being destroyed and recreated) before they are explicitly dismissed by the user. Depending on the cause, the lifecycle change may occur at either the *activity* or the *app* level.

Activity Restart. An activity restart can be triggered by *runtime configuration changes*. Table 2.1 lists the runtime configuration changes defined by the Android API (Level 25). For example, when the app window dimension is changed (e.g., due to a phone rotation or screen resizing), the system may decide to assign the activity a different layout that better matches the new dimension. Loading the new layout requires restarting the activity, going through the lifecycle from state *resumed* to *destroyed*, then back to *resumed* again (the red line in Figure 2.1). During the restart, a fresh activity instance is created and used for the subsequent user interaction. This process is known as *runtime change handling* [95]. Besides screen size changes, other runtime changes include changing the system language, attaching and detaching a keyboard, among others. All these runtime configuration changes by default result in activity restarting. Developers may overwrite the default behavior of runtime change handling, but it requires developers to manually load resources for the new configuration via system callback `onConfigurationChanged()`. According to our prior study [91], such customized handling is not common in practice.

App Relaunch. As mobile apps run on devices with limited resources, the system may run into low-memory situations, especially when the user has recently used memory-consuming apps [65]. When the memory pressure becomes high enough, the system will start to kill background apps by terminating their underlying Linux processes to reclaim memory [96]. When the app is killed, all the activities in a "task back stack" (recently visited, yet still active) will be destroyed first, before the app process is terminated. For this reason, when the user comes back to the killed app (e.g., from the "recent app list"), Android is responsible for relaunching it, which will create a fresh process for the app.

Table 2.2 summarizes the causes and consequences of these two levels of restarting. When an activity is restarted, the instance of the `Activity` (Java) class and the instances of all GUI elements specified in the layout file, are first destroyed, then recreated. When the app is relaunched, instances of the active activities are first destroyed, then recreated when the user comes back. In either case, it is critical that activity/app state is preserved,

Table 2.2: Restart Levels.

	Activity Level	**App Level**
Cause	Runtime configuration changes	High memory pressure
Effect	Instance of the current activity is destroyed	Instances of all active activities are destroyed

such that, from the user's perspective, it appears like no restarting or relaunching has ever happened.

2.2.1 Preserving App State

To facilitate app state preservation, Android provides two basic methods for developers to manage the app state during activity restart or app relaunch.

Saving/Restoring Instance State. Before the system stops an activity, it first invokes callback `onSaveInstanceState()` (see Figure 2.1) to give the activity a chance to save its state into a `Bundle` object. A `Bundle` is a persistent key-value map, serialized to disk, that survives app restarts or device reboots. To save data in the `Bundle` object, the data should be either primitive data (like `int`) or serializable objects (that implement `Serializable` or `Parcelable`). Note that, for GUI components with assigned IDs (either by `android:id` or `View.setId()`), Android automatically saves some of their user-editable properties (e.g., text in `EditText` or checking status of `RadioButton`). However, to save additional data, such as variables in the `Activity` class, properties of customized GUI elements, or non-user-editable GUI properties, developers need to override `onSaveInstanceState()` and add extra key-value pairs into the `Bundle` object to preserve them across activity/app lifecycles. When an activity instance is recreated or the app is relaunched, developers can recover the activity state by

Table 2.3: Two Basic Methods for Preserving App State.

	Saving Instance State	**ViewModel**
Storage location	Serialized to disk	In memory
Runtime change	Survives	Survives
System killing	Survives	Fails

extracting the data from the `Bundle` object, which is accessible in both `onCreate()` callback and `onRestoreInstanceState()` callback. Since the `Bundle` is persistent, the state saved via this mechanism can survive runtime configuration changes and system-initiated process kill.

ViewModel. `ViewModel`[69] is part of several newly released components for Android developers to manage UI-related data in a lifecycle-aware manner. Technically, the `ViewModel` is not part of the Android framework. A `ViewModel` can be created in association with an activity and will be retained in memory as long as the associated process is still live. Unlike saved instance state, a `ViewModel` can hold complex types of data without any serialization. Its in-memory saving solution works well for configuration changes, but cannot preserve data upon system-initiated killing.

In addition, developers can also leverage the fine-grained construct, `Fragment`, to retain some of the data during activity restart. Similar to `ViewModel`, `Fragments` cannot retain state in wake of system-initiated killing, where a fresh process of the app is created. Table 2.3 summarizes the two basic methods for preserving app state. Note that, in either method, developers need to first identify the data to preserve, then implement the preserving methods by either overwriting the callbacks to save and restore the app state or creating a `ViewModel` class that encapsulates the data. Both require a significant amount of programming effort to ensure the right set of data is preserved correctly. Unfortunately, as surveyed by recent work [91], a large majority of apps do not implement the state preserving methods appropriately. For example, 92.4% of activities allow restarting during runtime changes, but only 27.1% of activities implement one of the state preserving mechanisms. For simple activities, state loss upon restart may be small enough so re-creating it manually is not burdensome. However, as UI and app logic complexity increase, restarting an activity without sufficient data preservation makes the app vulnerable to various state issues. For instance, 172 state issues were reported in 72 popular apps [91]. In our evaluation (Section 3.4), we reveal 46 state issues found in 21 apps from Google Play store and GitHub, including highly popular apps. To free developers from this complex and error-prone task, this work proposes an automatic approach for identifying the app state that is necessary to preserve and tools for generating the state saving and restoring routines, together referred to as LiveDroid. Next, we first give an overview of LiveDroid.

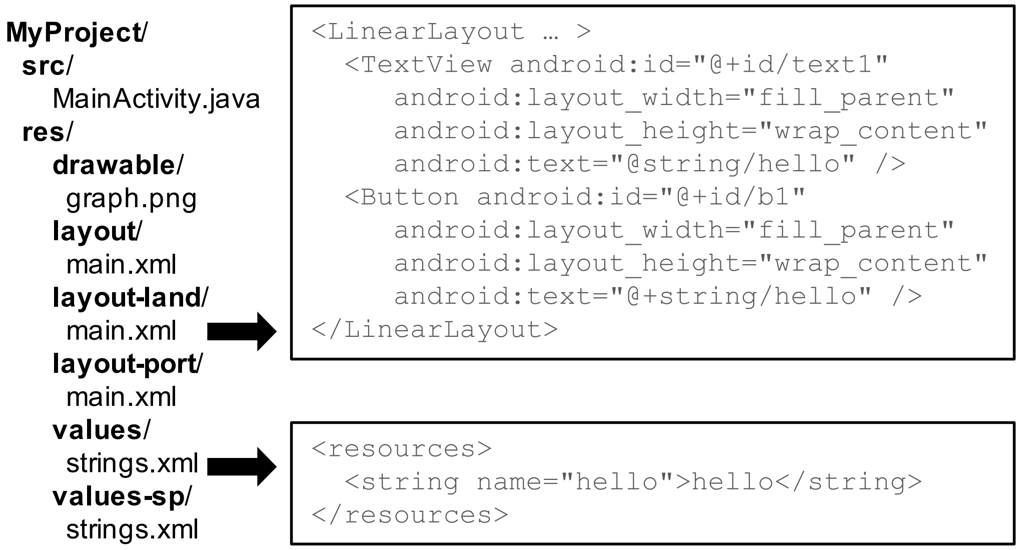
2.3 Runtime Change Handling

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 2.1 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 3.2).

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.



```
MyProject/
  src/
    MainActivity.java
  res/
    drawable/
      graph.png
    layout/
      main.xml
    layout-land/
      main.xml
    layout-port/
      main.xml
    values/
      strings.xml
    values-sp/
      strings.xml
```

```
<LinearLayout … >
  <TextView android:id="@+id/text1"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:text="@string/hello" />
  <Button android:id="@+id/b1"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:text="@+string/hello" />
</LinearLayout>
```

```
<resources>
  <string name="hello">hello</string>
</resources>
```

Figure 2.2: Example resources

Figure 2.2 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 [18].

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 3.3).

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 2.1).

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 [69]. 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.

2.4 App Testing and Profiling

Android offers several tools to test and measure performance of apps, including GUI testing using `dumpsys` [60], Espresso [55], Android Profiler [64], and App Crawler [58].

Profiling and Performance. `dumpsys` is a command line tool and offer several options to get information about system services. One of the option is `gfxinfo`, which provides graphics frame information, furthermore it can collect (using `framestats` option) UI performance data for the specific apps and outputs *janky* or delayed frames occurred during the execution.

GUI Testing. UI Exerciser Monkey [68] is UI stress testing tool, which sends random pseudo commands for user interaction events including device events as well. Another tool, UI Automator [67] helps to test cross-app UI in block-box style testing, it is useful to test whether app works correctly while interacting system and other installed apps. This tool can perform operations such as opening settings menu or launcher in target device. The Espresso [55] tests user interaction as other tools, and also provides deeper understanding about the app by allowing to test state expectations and assertions. To speed up testing, Espresso includes a test recorder [59] to record user interaction using an emulator or device and test recorder generates test cases code. However, state expectation and assertion remain up to developers and to *fully exploit Espresso features such as testing app state and assertions, developers need to be familiar with codebase.* More recently released App crawler tool [58] is part of JetPack [56], which automatically explores app by sending user interaction commands, such as click and swipe actions. As name suggests, this tool is more focused on finding and executing all possible app UI space, and do not allow to check app states or assertions.

Profiling Asynchronous Events. The Espresso provides profiling capabilities for Asynchronous events, it has built-in synchronization at `Message.Queue` level, which limits synchronization to Message Queue and AsyncTask constructs. Based on built-in synchronization, Espresso checks following conditions before continuing a UI test case.

1. The message queue is empty.
2. There are no instances of AsyncTask currently executing a task.
3. All developer-defined idling resources are idle.

Idling resource helps to overcome the limitation here, and developers can rely on idling resources to make sure subsequent UI updates have been completed. Idling resources work like Semaphores, when starting a background task increment and when finished decrement [61]. However, developers are *required to add code to idle resources during testing and remove before the release.*

2.5 Finding Bugs for Mobile Apps

There exist a large body of work in detecting bugs for mobile apps. AppDoctor [105] 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 [162] 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 [45] inject *neutral* event sequences, such as pause-resume, pause-stop-restart and pause-stop-destroy-create to test apps. Shan and others [142] 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 [49] 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 [103, 127, 104], uncovering bugs with network and location data [120], detecting performance bugs [125, 121, 128, 80, 75] and memory leaks [161].

2.6 API Usage and Mobile App Refactoring

There are also empirical studies on programming languages and libraries usages [83, 85, 88, 100, 110, 131]. Buse and others [83] introduce an automatic technique for mining and synthesizing documents for program interfaces. Kavaler and others [111] investigate Android APIs questions on Stackoverflow [148]. Unlike prior work, this work studies the APIs and practices of runtime change handling. Bavota and other [79] 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 [141], refactoring global state with thread-local state [140], refactoring the concurrent programming constructs [129], and refactoring for energy efficiency [137]. Unlike the objectives of prior work, this work aims for a refactoring-based solution for addressing issues in runtime change mishandling.

2.7 Management of State in Mobile Apps

Shan et al. [142] used program analysis to discover Kill and Restart (KR) errors in Android apps; they combined static and dynamic analysis to verify KR errors. In contrast, our work focuses on identifying critical app data that should be preserved during the app restarting, followed by an automatic save/restore solution. More recently, Lebeck et al. [114] proposed a new memory manager for Android, which swaps the apps to the external storage instead of killing them to reclaim memory when memory runs low. However, enabling disk swapping may shorten the lifespan of the flash drive [73]. Moreover, activity will still be restarted during runtime configuration changes. In comparison, our work is capable of handling app killing and activity restarting by identifying and preserving necessary instance states.

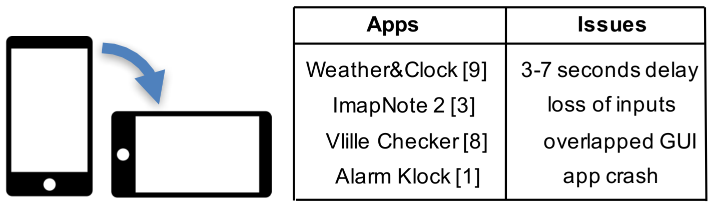
Chapter 3

Restarting-Free Runtime

3.1 Introduction

Nowadays, smartphone, tablets, and wearable devices are emerging as an essential component of modern life. According to IDC [19], 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 [136]. For multilingual users, changing the language setting is often needed [37] and for tablet users, attaching an external keyboard often ease the uses of tablets [39]. As newer versions of



Apps	Issues
Weather&Clock [9]	3-7 seconds delay
ImapNote 2 [3]	loss of inputs
Vlille Checker [8]	overlapped GUI
Alarm Klock [1]	app crash

Figure 3.1: An example runtime change and its issues.

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 [34].

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 3.1 lists four example issues [41, 31, 40, 21] 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 [142] 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 [23, 24, 25, 22].

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 present the formative study on runtime change handling, including its common issues (Section 3.2). After that, we will present RUNTIME-DROID and its implementations (Section 3.3), followed by the evaluation (Section 3.4), conclude this work (3.5).

3.2 Formative Study

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. 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.

3.2.1 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 [94], 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 [13], K-9 email [32], Google I/O [9], Amaze File Manager [11], Timber player [12], and Wordpress [43], just to name a few. Many of these apps are hosted on Google Play Store [99]. 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 (Chapter 2), RuntimeAnalyzer also checks the uses of 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.

Table 3.1: 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 3.2: 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%)

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 3.1-3.3 report the statistical results of the study. As shown in Table 3.1, 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

Table 3.3: 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%)

activity without sufficient data preservation makes the app vulnerable to various runtime change issues (see Section 3.2.2).

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 3.4 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 3.3, 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, 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

Table 3.4: 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%)

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.

3.2.2 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 3.2.1). 36 out of the 72 apps are also hosted on Google Play Store [99], including quite a few highly popular ones, such as *Loop - Habit Tracker* [10] with 1M installs, *WiFiAnalyzer* [15] with 1M installs, Barcode Scanner [8] 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 3.1) 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 2.1). 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` [40], 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.

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 3.2 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

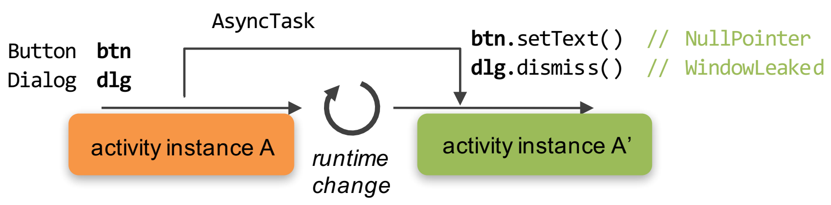


Figure 3.2: A common app crash scenario, caused by the misuse AsyncTask during activity restarting.

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.

3.3 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.

3.3.1 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.

3.3.2 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 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.

$$RH_{\text{diff}} = RH_{\text{old}} - RH_{\text{new}} \tag{3.1}$$

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

M_{PR} :

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

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 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 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 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 Chapter 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.

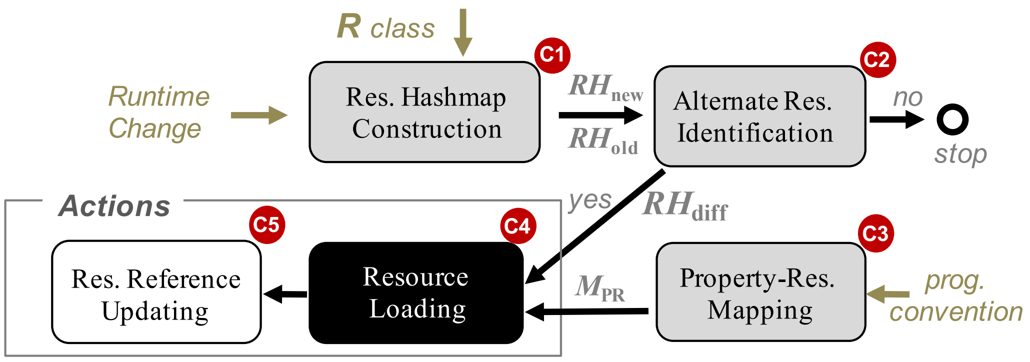


Figure 3.3: High-level Workflow of HotR.

Summary. Figure 3.3 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.

3.3.3 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 *A* <u>static view hierarchy</u>*, denoted as $\mathcal{V}$, is the initial view hierarchy that is derived from the activity layout (XML file).*

Definition 5 *A* <u>dynamic view hierarchy</u>*, 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' 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 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 Dynamic View Hierarchy Migration

```
1: $\mathcal{V}_{\text{old}}$: the static view hierarchy before runtime change
2: $\widetilde{\mathcal{V}}_{\text{old}}$: the dynamic view hierarchy before runtime change
3: $\mathcal{V}_{\text{new}}$: the static view hierarchy after runtime change
4: $\widetilde{\mathcal{V}}_{\text{new}}$: the dynamic view hierarchy after runtime change
5: /* preparation */
6: $V_{\text{dyn}}$ = compare($\mathcal{V}_{\text{old}}$, $\widetilde{\mathcal{V}}_{\text{old}}$);    ▷ identify dynamic views
7: $V_{\text{static}}$ = derive($\mathcal{V}_{\text{old}}$);    ▷ derive static views
8: $P_{\text{mut}}$ = find(Activity.class);    ▷ identify mutable properties
9: /* migrate static views */
10: **for** each view $v$ in $V_{\text{static}}$ **do**
11:     **for** each property $p$ of $v$ **do**
12:         **if** $p \in P_{\text{mut}}$ **then**
13:             **if** $p$ has $RH_{\text{diff}}$ **then**    ▷ $RH_{\text{diff}}$: res. w/ alternatives
14:                 loadResource($p$, $RH_{\text{diff}}$);
15:             copy($p$, $\mathcal{V}_{\text{new}}(v)$);    ▷ $\mathcal{V}_{\text{new}}(v)$ returns the view w/ same id
16: /* migrate dynamic views */
17: **for** each view $v$ in $V_{\text{dyn}}$ **do**
18:     **if** $v$ was deleted from $\mathcal{V}_{\text{old}}$ **then**
19:         detach($v$, $\mathcal{V}_{\text{new}}$);    ▷ detach $v$ from $\mathcal{V}_{\text{new}}$
20:     **else**    ▷ $v$ was attached to the $\widetilde{\mathcal{V}}_{\text{old}}$
21:         **for** each property $p$ of $v$ **do**
22:             **if** $p \in P_{\text{mut}}$ && $p$ has $RH_{\text{diff}}$ **then**
23:                 loadResource($p$, $RH_{\text{diff}}$);
24:         attach($v$, $\mathcal{V}_{\text{new}}$);    ▷ attach $v$ to $\mathcal{V}_{\text{new}}$
25: $\widetilde{\mathcal{V}}_{\text{new}}$ = $\mathcal{V}_{\text{new}}$;    ▷ after updating $\mathcal{V}_{\text{new}}$, it becomes $\widetilde{\mathcal{V}}_{\text{new}}$
26: **return** $\widetilde{\mathcal{V}}_{\text{new}}$;
```

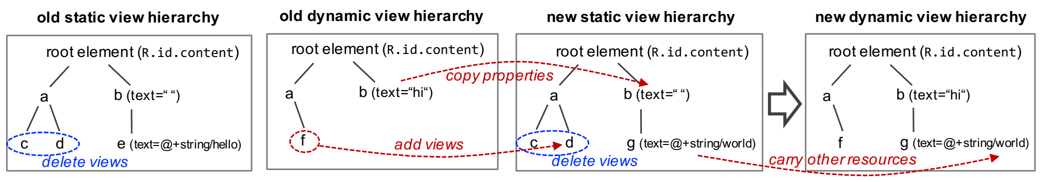

old static view hierarchy
root element (R.id.content)
a
b (text=" ")
c
d
e (text=@+string/hello)
delete views
old dynamic view hierarchy
root element (R.id.content)
a
b (text="hi")
f
add views
copy properties
new static view hierarchy
root element (R.id.content)
a
b (text=" ")
c
d
g (text=@+string/world)
delete views
carry other resources
new dynamic view hierarchy
root element (R.id.content)
a
b (text="hi")
f
g (text=@+string/world)

Figure 3.4: Illustration of Dynamic View Hierarchy Migration.

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 3.4 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).

3.3.4 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 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 [144] and APKtool [20] for reverse engineering and recompilation, and zipalign [44] and Jarsigner [17] to align and sign the processed APK file. The key step – code refactoring – was implemented by ourselves.

Table 3.5: Results of applying RuntimeDroid to the activities in Android projects – 1/2. A: # of activities, A_s: # of activities successfully processed, str: # of string resources, V: number of static views, I: # of issues, I_f: # of fixed issues, A_l: # of activities requiring resource reference localization

#	App Project	A	A_s	str	V	I	I_f	A_l
1	0xbb/otp-authenticator	2	1	14	7	2	2	1
2	Amabyte/vtu-cs-lab-manual	5	5	3	26	1	1	3
3	AntennaPod/AntennaPod	19	19	115	109	5	5	14
4	arnowelzel/periodical	5	5	61	141	2	2	4
5	artemnikitin/tts-test-app	1	1	4	9	1	1	1
6	awaken/sanity	28	28	147	8	9	9	18
7	balau/fakedawn	3	3	1	39	2	2	3
8	basil2style/getid	2	2	0	34	1	1	2
9	benjaminaigner/aiproute	3	3	28	20	3	3	2
10	blanyal/Remindly	4	4	23	83	2	2	4
11	blaztriglav/did-i	2	2	6	9	3	3	2
12	cbeyls/fosdem-companion-android	8	8	21	24	5	5	7
13	charbgr/Anagram-Solver	1	1	0	1	2	2	1
14	charlieCollins/and-bookworm	10	10	113	81	4	4	9
15	conchyliculture/wikipoff	8	8	58	79	2	2	7
16	DF1E/SimpleExplorer	4	4	6	14	1	1	3
17	enricocid/Color-picker-library	2	2	4	8	1	1	2
18	erickok/transdroid-search	3	3	0	0	1	1	2
19	EvanRespaut/Equate	2	2	18	3	2	2	2
20	farmerbb/Taskbar	24	24	65	22	3	3	9
21	fr3ts0n/StageFever	2	2	1	3	1	1	2
22	gateship-one/malp	5	5	34	51	5	5	4
23	gateship-one/odyssey	4	4	67	25	4	4	3
24	gianluca-nitti/android-expr-eval	2	2	14	14	1	1	2
25	google/google-authenticator-android	12	12	85	50	3	3	11
26	grmpl/StepandHeightcounter	2	2	20	5	2	2	2
27	grzegorznittner/chanu	22	22	114	105	2	2	9
28	hoihei/Silectric	5	5	9	38	2	2	5
29	HoraApps/LeafPic	9	9	206	188	2	2	8
30	HugoGresse/Anecdote	1	1	7	3	1	1	1
31	icasdri/Mather	2	2	5	5	1	1	2
32	iSoron/uhabits	8	7	6	5	6	6	1
33	JamesFrost/SimpleDo	6	6	26	70	6	6	6
34	jiro-aqua/aGrep	6	6	31	25	3	3	5
35	jparkie/Aizoban	4	4	26	21	1	1	4
36	jpriebe/hotdeath	3	3	5	5	5	5	3

Table 3.6: Results of applying RuntimeDroid to the activities in Android projects – 2/2. A: # of activities, A_s: # of activities successfully processed, str: # of string resources, V: number of static views, I: # of issues, I_f: # of fixed issues, A_l: # of activities requiring resource reference localization

#	App Project	A	A_s	str	V	I	I_f	A_l
37	jufickel/rdt	1	1	10	25	1	1	1
38	julian-klode/dns66	3	3	27	21	1	1	3
39	knirirr/BeeCount	8	8	85	53	2	2	8
40	kraigs-android/kraigsandroid	2	2	15	17	5	5	2
41	liato/android-bankdroid	12	12	69	97	3	3	9
42	LonamiWebs/Stringlate	10	10	117	73	5	5	9
43	mikifus/padland	10	10	59	33	3	3	5
44	nathan-osman/chronosnap	2	2	16	16	1	1	2
45	nbenm/ImapNote2	4	4	0	37	3	3	3
46	netmackan/ATimeTracker	5	5	70	21	15	15	4
47	ojacquemart/vlilleChecker	5	4	9	41	1	1	1
48	olejon/mdapp	42	42	392	284	2	2	39
49	PaperAirplane-Dev-Team/GigaGet	4	4	13	44	1	1	4
50	peoxnen/GitHubPresenter	1	1	1	3	2	2	1
51	phikal/ReGeX	4	4	33	54	3	3	4
52	phora/AeonDroid	5	5	0	44	4	4	2
53	phora/AndroPTPB	6	6	0	43	1	1	4
54	pilot51/voicenotify	2	2	41	5	1	1	1
55	quaap/Primary	11	11	52	67	4	4	10
56	RomanGolovanov/ametro	6	6	22	34	1	1	6
57	rubenwardy/mtmods4android	7	7	54	51	1	1	6
58	scoute-dich/PDFCreator	8	8	32	22	4	4	7
59	scoute-dich/Sieben	32	32	677	112	2	2	30
60	scoute-dich/Weather	8	8	108	33	3	3	8
61	SecUSo/privacy-friendly-ruler	6	6	22	47	2	2	4
62	shkcodes/Lyrically	2	2	11	21	2	2	2
63	SteamGifts/SteamGifts	11	10	20	33	3	3	7
64	tarunisrani/InstaHack	2	2	0	13	1	1	2
65	TeamNewPipe/NewPipe	13	13	39	74	6	6	7
66	TobiasBielefeld/Simple-Solitaire	6	6	27	48	1	1	5
67	ukanth/afwall	14	14	248	101	4	4	13
68	vIiRuS/Omnomagon	4	4	32	71	1	1	2
69	VREMSoftwareDevelopment/WiFiAnalyzer	3	3	20	36	2	2	3
70	wentam/DefCol	5	5	0	18	1	1	5
71	xargsgrep/PortKnocker	5	5	24	15	4	4	4
72	zxing/zxing	9	9	41	50	4	4	6

3.4 Evaluation

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

3.4.1 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[99], including some highly popular ones (see Section 3.2.2 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.

3.4.2 Applicability

Tables 3.5 and 3.6 summarize 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, Tables 3.5 and 3.6 also report 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 Tables 3.5 and 3.6 indicate that a large ratio of activities (86%) require resource reference localization, due to global-level declarations of certain resources (see Section 4.2).

3.4.3 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 3.2.2), 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 Chapter 2) to fix the issues. However, these solutions 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.

3.4.4 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 4.7 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

Table 3.7: 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

`weather&clock` [41](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 [14]. 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.

3.4.5 Time and Space Costs

Time Costs. Table 4.7 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

Table 3.8: 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

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.

Space Costs. Table 4.8 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 3.3.4).

3.5 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.

Chapter 4

Identifying and Preserving State

4.1 Introduction

Smartphones are in wide use (there were 3.2 billion smartphone users worldwide in 2019 [150]) and mobile apps have a substantial economic impact (the mobile app market is projected to reach \$407 billion by 2026 [48]). Hence, there is an impetus for ensuring and improving mobile app reliability. Building reliable mobile apps poses additional complications when compared to desktop/server applications, due to the challenges imposed by *rich yet volatile* mobile runtime environments.

Volatile Runtime Environment. Unlike desktop or server applications, mobile apps run in a more challenging environment: devices are resource-limited, and the underlying OS subjects the app to a richer set of disruptive events. Consequently, mobile apps often go through multiple *lifecycles* – being destroyed and recreated – before they are explicitly dismissed. For Android apps, when a runtime configuration change occurs, like a phone rotation (portrait $\leftrightarrow$ landscape) or attaching a keyboard, the OS destroys the current screen instance (a.k.a *activity* in Android), including both the GUI elements and (Java) class associated with the screen, and then recreates a new screen instance. This process is known as *activity restarting.* The purpose of activity restarting is to automatically reload the activity with resources that match the new configuration (e.g., landscape mode layout after rotation) [95]. Another destructive scenario involves low resources: a running app (especially when sent to the background) can be killed at any time by the OS when memory runs low, then relaunched when the user comes back to the app [65, 74]. This is due to mobile OSes, including both iOS and Android, eschewing swapping (i.e., paging out) [65, 70], to minimize flash memory wear [73]. When an app is killed due to low memory, all its running activities are destroyed.

To avoid losing user progress, or entering into an inconsistent state, certain program variables and properties of the GUI elements must be saved before the activity is destroyed (or app is killed) and restored after the activity gets recreated (or app gets relaunched), as if the activity (or app) remains running in the same lifecycle [96, 71]. We refer to this set of data, that is *necessary* to preserve in order to maintain the illusion that the activity or app is always running, as *necessary instance state.*

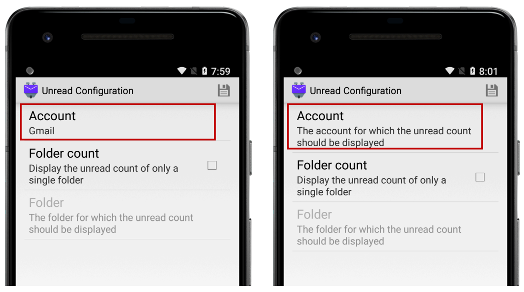
State of The Art. Currently, while the Android system saves and reinstates some GUI state upon restart, developers still have to explicitly perform a substantial amount of data saving and restoring using system callbacks upon activity restarts [97]. For many real-world apps, it is non-trivial to manually reason about necessary instance state, as it depends on how user interaction and system events affect the program variables and GUI properties, which is loosely defined in various callbacks. This challenge is further compounded by the

Table 4.1: Example State Issues Triggered by Volatile Runtime Environments.

GitHub Repo	Issue ID	Issue Description
WordPress [43]	12223	Rotate the device (either from portrait to landscape or vice versa) while the tag editor is open in Post Settings, the tags are removed.
K9 Mail [32]	4519	Open *General settings*, click on the search icon, enter some text, then change screen orientation, the app crashes.
K9 Mail [32]	4936	Create *Unread widget*, click on *Account* and select user mail-box, then rotate the device, the selected account is lost.
OpenMF [35]	829	Run the app, generate a collection sheet, select office from the drop-down, change the orientation. The state gets refreshed.
TileView [38]	535	Put `TileView` in layout, set layout as content view in `Activity`, put the app to background, kill the process (either from system or logcat), then bring the app to foreground, the app crashes.
MapBox [33]	3517	Open *Press for marker* activity, long press to add a marker, click the marker to open infowindow, then rotate the device (infowindow is closed; marker is visible), clicking marker results in a crash.
Glucosio [30]	431	Open the app, fill user information such as language, gender and age, then change the orientation by rotating, the user inputs are lost.

complex lifecycle stage transitions. For example, a StackOverFlow question [36] on how to deal with initialization in the presence of activity restarting received 1394 thumbs-up.[1] Recent studies [142, 91] have shown that when handled improperly, Android apps may suffer from various runtime issues, ranging from data loss to unresponsiveness, UI distortion, and app crashes. In this work, we refer to these runtime issues that are caused by failing to save and restore the data in the necessary instance state as *state issues*. Table 4.1 lists a few example state issues found in several very popular GitHub repositories, including K-9 Mail [32], MapBox [33], WordPress [43], TileView [38], OpenMF [35], and Glucosio [30].

[1]As of September 9th, 2020.


Unread Configuration

Account
Gmail

Folder count
Display the unread count of only a single folder

Folder
The folder for which the unread count should be displayed

Before

Unread Configuration

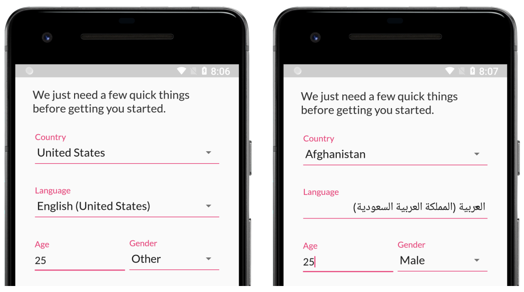
Account
The account for which the unread count should be displayed

Folder count
Display the unread count of only a single folder

Folder
The folder for which the unread count should be displayed

After

(a) Lost account binding in K-9 Mail [32]


We just need a few quick things before getting you started.

Country: United States
Language: English (United States)
Age: 25
Gender: Other

Before

We just need a few quick things before getting you started.

Country: Afghanistan
Language: العربية (المملكة العربية السعودية)
Age: 25
Gender: Male

After

(b) Lost user inputs in Glucosio [30]

Figure 4.1: Example State Issues of Two Popular Android Apps.

Take K-9 Mail and Glucosio as examples: after a configuration change (e.g., screen rotation), K-9 loses its binding to the Gmail account, as shown in Figure 4.1-(a); Glucosio loses user inputs, such as country, language, and gender, as shown in Figure 4.1-(b). Such unexpected app behavior negatively affects user experience.

To mitigate the aforementioned challenges, prior efforts have focused on either detecting [142] or preventing [91] state inconsistency. Shan et. al [142] focus on detecting the *control-flow disparity* in saving and restoring of *mutable activity fields* – whether a conditionally saved variable is restored under the same condition, and vice versa. However, as we will show later, not all mutable activity fields are part of the necessary instance state. Serious over-saving may lead to observable delays that negatively impact the user experience, as both saving and restoring usually occur while the user interacts with the app (e.g., during a phone rotation). Additionally, Shan et al.'s work does not take into account GUI elements declared in resource files, which may also carry the past user interaction. In contrast, The approach in RuntimeDroid̃citeruntimeDroid is to prevent the activity from restarting at

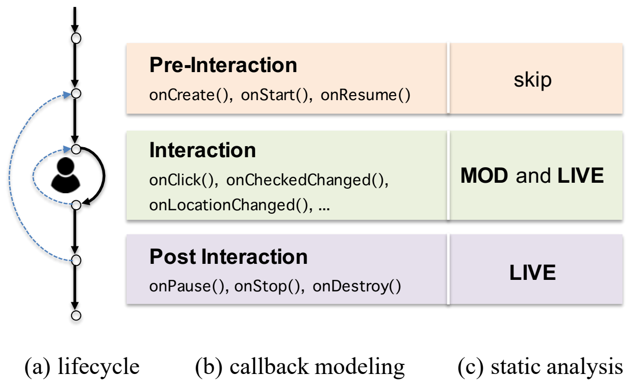


Figure 4.2: Callback Modeling and Static Analysis.

all, by overwriting the default configuration change handling, thus eliminating the needs for data saving and restoring. However, this approach cannot handle system-initiated app killing – activities will still be forced to restart once memory runs low, in which case apps can still lose their states. More critically, no prior work has systematically addressed the fundamental question – *how to identify the necessary instance state of mobile apps*?

Overview of This Work. The goal of this work is to leverage static analysis to answer the above question – *statically identifying the necessary instance state of mobile apps* and *automatically generating the state saving and restoring routines*, thus freeing developers from this tedious and error-prone task. Similar to prior work, we focus this work on the Android platform due to the platform's popularity (75% market share as of July 2020 [149]) and open-source ecosystem. To achieve our goal, we propose (i) a three-phase model to characterize the callbacks based on their potential impacts on app state, (ii) a combination of static analyses that reason about the app source code and resource files to identify program variables and GUI properties that belong to the necessary instance state, and (iii) tools for developers to generate state-saving/restoring routines.

Android apps consist of many different callbacks (i.e., event handlers) to respond to various events, including those generated from user interactions (e.g., clicking, scrolling, and typing), as well as those triggered by system updates (e.g., battery status and location changes). To capture the impact of different callbacks on app state, we break down the activity lifecycle into three phases – (i) *pre-interaction*, (ii) *interaction*, and (iii) *post-interaction* – and group the callbacks accordingly, as shown in Figure 4.2-(a-b). Then, for callbacks of different categories, we perform a suite of analyses to find out which access paths of variables (like `this.account.user.addr`) and properties of GUI elements (like `this.mEditText.text`) are part of the necessary instance state. Informally, there are two basic requirements for an access path to be in the necessary instance state:

- ***Live***: Any future "use" of the access path after the activity restart or app relaunch should yield the same result as if the the restart or relaunch had not happened;
- ***Modified***: The access path should be *modified* (written) at least once by a callback during the interaction phase (note that this excludes the initialization in the pre-interaction phase).

The first requirement captures the fact that the necessary instance state to be preserved must be *sufficient* to guarantee *correctness*, by including all the access paths that may be used (i.e., *live*) for future interactions. To fulfill this requirement, we conduct an *interprocedural entry-liveness analysis* on callbacks belonging to the interaction and post-interaction phases. We exclude the pre-interaction phase as it belongs to "the past" – activity restart or app relaunch only happens after this phase. Let the result of this analysis be *LIVE*.

The second requirement reveals the fact that the necessary instance state should reflect the past user interaction and system updates. It excludes the access paths that cannot be modified during the interaction, or callbacks invoked before or after the interaction. Since they either remain unchanged after the activity restart or app relaunch, or carry no effects of user interaction or system updates. To fulfill this requirement, we perform an *interprocedural may-modify analysis* on callbacks belonging to the *interaction* phase. Let the result of this analysis be *MOD*.

Finally, we take the intersection between *LIVE* and *MOD* to obtain a static over-approximation of the internal[2] necessary instance state, denoted as $NISTATE_{in}$ (i.e., $NISTATE_{in} = LIVE \cap MOD$). Note that even though we leverage a series of techniques to improve the precision of the static analyses, including *field-sensitivity* and *alias-awareness*, over-approximation in general is often unavoidable due to the nature of static analysis. Besides analyzing the activity (Java) classes, we also perform a *UI property analysis* on activity resource files to find out the external necessary instance state $NISTATE_{ex}$ regarding the GUI elements that are directly visible to users. The design of UI property analysis also follows the two basic requirements (*live* and *modified*). Putting them together, we have the necessary instance state $NISTATE = NISTATE_{in} \cup NISTATE_{ex}$.

Based on the formalization above, we implemented our static analyzer, on top of the Soot analysis framework [144]. Furthermore, to facilitate the use of the analysis results, we also designed and developed: (i) an Android Studio plugin that interactively guides developers to generate the state-saving and restoring routines, and (ii) an APK patching

[2]It is internal in the sense that they are not GUI elements, though they may include GUI element references.

tool that automatically inserts state-saving and restoring routines into the app binary code to preserve the *NISTATE*. Together, we refer to the entire app state handling solution as LiveDroid.

We evaluated LiveDroid on both a large corpus of 966 apps and a focused corpus of 36 apps collected from F-Droid [89], Google Play [99], and GitHub. The evaluation shows that LiveDroid can be successfully applied to the apps in the large corpus, and can correctly and precisely identify the necessary instance states of apps in the focused corpus. On one hand, compared to the state-of-the-art app state identification approach [142] which includes all mutable fields of the activity but ignores the GUI properties, LiveDroid yields much smaller app states to preserve, decreasing the delay for state saving and restoring by 16.6X (1.7X - 141.1X) and 9.5X (1.1X - 43.8X), respectively. On the other hand, compared to manual state handling performed by developers, the static analysis of LiveDroid reveals a set of 46 app state issues due to insufficient state saving/restoring, all of which can be successfully eliminated after applying LiveDroid. Artifacts related to this evaluation are available via `https://github.com/ucr-riple/LiveDroid`.

In summary, this work makes the following contributions:

- We introduce *necessary instance state* based on liveness and modification to capture the essential data that need to be preserved during activity restarting and app relaunching.
- We model and categorize the callbacks based on their invocation orders relatively to the user interaction such that their impacts on the app state can be more precisely analyzed.

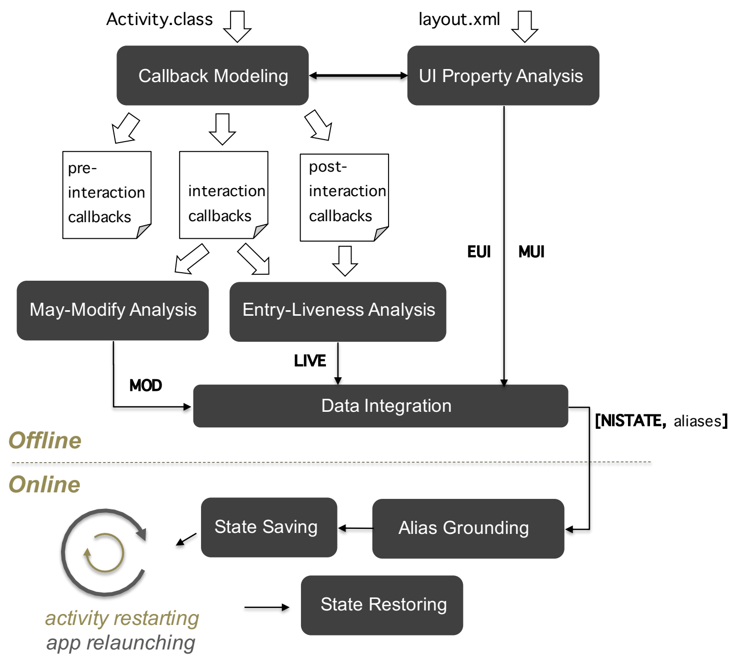


Figure 4.3: Overview of LiveDroid.

- We present static analyses (inter-procedural entry-liveness, may-modify, etc.) to automatically compute the necessary instance state for a given app.
- To handle aliasing, we combine points-to analysis, an access path abstraction, and dynamic checking to achieve the required precision and scalability.

4.2 Overview

Figure 4.3 shows LiveDroid's architecture. At the high level, it follows a *hybrid* design consisting a *static analyzer* (the upper part) and a *runtime module* (the lower part). The static analysis is applied to each app activity offline (only once) to identify the necessary instance state (*NISTATE*); the runtime part verifies certain properties of *NISTATE* and performs saving and restoring. There are two main reasons for this hybrid design. First, the aliasing relationships among references may change at runtime; it is impossible for a static analysis to determine them. As we will show later, failing to preserve the exact aliasing relationship may compromise correctness. On the other hand, a purely runtime solution that tracks the actually changed state could minimize the necessary instance state, but requires monitoring every update to the entire app state, which may only work well for apps with small-sized app state and limited dynamic features.

In the first component of the static analyzer, *callback modeling*, all the registered callbacks in the activity are grouped into three basic categories based on the phases in which they may occur: *pre-interaction callbacks*, *interaction callbacks*, and *post-interaction callbacks*. Note that the interaction callbacks include both the system callbacks and the UI callbacks (see Chapter 2). The categorized callbacks (except those in the first category) are then fed into two major static analysis components: *entry-liveness analysis* and *may-modify analysis*. The former reports the access paths of the activity that are *live* (i.e., used before they are defined) at the callback entries (i.e., *LIVE*). The latter identifies the access paths of the activity that may be modified during the interaction phase (i.e., *MOD*). Note that the above analyses only capture "internal" state in Java code but not the "external" state – properties of GUI components declared in the layout files ("resources" in Android parlance). The layout files of the activity are fed into the *UI property analysis* component, which extracts editable properties of declared GUI components (i.e., *EUI*) and also properties that may be modified by the interaction callbacks (i.e., *MUI*). Finally, results of the above analyses are integrated by the *data integration* component and the necessary instance state, *NISTATE*, is produced. To ensure correctness in the presence of reference comparisons and to avoid duplicate object saving and restoring, the data integration also yield a set of statically found *aliases*.

The runtime module has three components which run with the app; they are integrated as parts of the app through code generation. The first component, *alias grounding*, checks which statically-identified aliases are the actual aliases when the activity is destroyed, such that only these actual ones are preserved (for correctness purposes). The alias grounding can also ensure that the object pointed to by one alias class is saved and restored only once. Next, the *state saving* module saves the access paths in the *NISTATE*. For references, we first serialize the corresponding objects, then add them along with the primitive access paths into the `Bundle` object. These first two components are automatically invoked by `onSaveInstanceState()` before the activity or the app gets destroyed. Once the restarting/re-launching process is completed, the third component, *state restoring* is invoked by callback `onRestoreInstanceState()`, which extracts data from the `Bundle` object and deserializes it into the corresponding access paths. One complexity in the design of the runtime module lies in the handling of private and partial objects, which we will address in Section 4.4.

In the following two sections, we describe these modules in detail.

4.3 Static Analyses

Given an Android activity, the *domain* of our static analyses include all the *access paths* in the activity class (a Java class) and the *access paths* in the GUI elements declared in the layout files. Here, an access path is *a sequence of fields rooted in the activity class* (e.g., `Activity.user.name`) *or rooted in a GUI element instance declared in one of the layout files for this activity* (e.g., `TextViewId.text`). The goal of the static analyses is to discover those access paths that must be preserved to make activity restarts and app relaunches *transparent* to the user. To achieve this, we next discuss two key access path properties: *liveness* and *modification.*

- ***Liveness.*** First, for correctness, after an activity restart or app relaunch, *any "use" of an access path should yield the same result as it would have without the restart or relaunch.* The set of access paths that our technique automatically saves and restores must be *sufficient* to guarantee this property. Note that if an access path is always defined (written) within a callback before it is used (read), there is no need to preserve the access path. Hence, following the data-flow analysis terminology, an access path must be *live* right after the activity restart or app relaunch to be a candidate for saving and restoring.

- ***Modification.*** Second, from another perspective, if any "use" of an access path is already guaranteed to yield the same result before and after an activity restart or app relaunch, there is no need to save and restore the access path. In other words, we only need to save and restore the access paths that may be modified by the user interaction or system updates.

The above two properties form the core design principles for our approach. Based on them, we design three static analyses: *entry-liveness analysis*, *may-modify analysis*, and *UI property analysis.* The first two are for the "internal" access paths defined in the `Activity` class while the third is for the "external" access paths – the GUI elements and their properties declared in the layout files (and `Activity` class[3]). To find out the appropriate targets (codes) for each static analysis, we model the callbacks, including all UI, system, and lifecycle callbacks, based on their timing constraints and impacts on the app state. Next, we first present the callback modeling.

4.3.1 Callback Modeling

Note that *specific callbacks are attached to specific lifecycle stages of an activity*, which leads to different impacts on the activity state. For example, UI and system callbacks can only be invoked when the activity is in the *resumed* stage. Before that, the callbacks mainly initialize the activity and allocate resources. Based on this observation, we partition the activity lifecycle into three phases based on the availability of the activity for user and system interaction: (i) *pre-interaction phase*, (ii) *interaction phase*, and (iii) *post-interaction phase*. Then, according to the phases where the callbacks may be invoked, we group the callbacks of each activity into three categories:

Definition 7 Pre-interaction callbacks *invoked before the activity becomes available for user and system interaction (i.e.,* resumed*), including `onCreate()`, `onStart()`, and `onResume()`.*

Definition 8 Interaction callbacks *that may be invoked after the activity becomes ready for user and system interaction – the activity is* resumed*. These include all registered UI and system callbacks, such as `onClick()` and `onLocationChanged()` (among others).*

Definition 9 Post-interaction callbacks *executed when the activity is no longer available for user and system interaction, including `onPause()`, `onStop()`, and `onDestroy()`.*

[3]A GUI element can also be dynamically declared in Java class; more details will be given later.

Besides the above, there is also one lifecycle callback that we deliberately ignore – `onRestart()`, which is used for handling restarts. Since our goal is to automate the restart handling process, there is no need to include it in the following analyses. In addition, there are also callbacks related to asynchronous tasks (i.e., `AsyncTask` [52]), which can be launched in the interaction phase. Hence, these callbacks also belong to the interaction callbacks. However, they are treated slightly differently than others due to the unrecoverable nature of asynchronous tasks (more details in Section 4.4). Specifically, LiveDroid only analyzes the `onPostExecute()` callback, triggered when the asynchronous task is fully completed. Other related callbacks, such as `onProgressUpdate()` (for handling progress updates), will not be analyzed for state saving/restoring under the assumption that their impacts on the activity instance state are temporary and can be recreated when the asynchronous task is relaunched (see Section 4.4). Based on the callback modeling, the analyses will become more focused, as we will show next.

4.3.2 Entry-Liveness Analysis

According to the liveness property, we need to find all the access paths that are live right after activity restart or app relaunch. Because that is the moment when the activity comes back to the *resumed* stage, only interaction and post-interaction callbacks may access them in the future.[4] Therefore, we only need to perform static analysis on these callbacks. Moreover, as an activity is never restarted (and an app is never relaunched) in the middle of a callback execution, we only need to find the access paths that are live *at the entry* of the

[4]Pre-interaction callbacks execute during a restart before instance state is restored (`onRestoreInstanceState()` in Figure 2.1), hence they cannot rely on saved instance state.

callbacks, hence the name *entry-liveness analysis*. Next, we formally define the concept of liveness, then present its analysis.

Liveness is a well-known compiler concept that has been used for register allocation [46], garbage collection [47], etc. In this work, we use it for identifying the app state to preserve. Formally, the liveness of a variable can be defined as follows.

Definition 10 *A variable v is* live *at program point p, if and only if there is an execution path from p to a use of v, along which v is not redefined.*

Code 4.1 lists the *LIVE* set for each statement in the callback `onClick()`, which contains the access paths that are live right before the statement. For example, at line 7, *LIVE* = {`this.tView`, `this.d`, `this.dView`}, because access paths `this.tView` and `this.dView` will be used at line 12 and 13 and access path `this.d` will be used in line 8. Note that though `this.t` will also be used in line 8, it will be first redefined (killed) in line 7. Therefore, it is not live right before line 7.

As a classical analysis, liveness analysis is typically solved *iteratively backwards* [46]. Initially, at the exit of a callback, no access path is live (see line 9 in Code 4.1). As the analysis traverses backwards, depending on whether the statement is a reference copy statement (like the one in line 4), different rules (i.e., transfer functions) are applied. For a non-reference-copy statement, the analysis first removes the access paths it defines/kills (denoted as *DEF*) from *LIVE*, then adds the access paths it uses (denoted as USE) to *LIVE*. For a reference copy statement, if its left-hand side reference (LHS) appears as the prefix of some access paths in *LIVE*, we substitute the prefix with its right-hand side reference (RHS). At line 4, we substitute the prefix of `this.f.b` with `this.a.f`. As a result, `this.a.f.b`

```java
class FooActivity extends Activity{
  ...
  void onClick() {
    this.f = this.a.f; //LIVE = {this.a.f.b, this.tView, this.d, this.dView}
    if(this.f.b == true) //LIVE = {this.f.b, this.tView, this.d, this.dView}
      this.d = this.d * 9 / 5 + 32; //LIVE = {this.tView, this.d, this.dView}
    this.t = Calendar.getInstance().getTime(); //LIVE = {this.tView, this.d,
        this.dView}
    updateViews(this.t, this.d); //LIVE = {this.t, this.tView, this.d, this.dView}
    //LIVE = { }
  }
  void updateViews(Time t, Degree d) {
    this.tView.setText(t); //LIVE = {t, this.tView, d, this.dView}
    this.dView.setText(d); //LIVE = {d, this.dView}
  }
}
```

Code 4.1: Example of liveness analysis.

replaces `this.f.b` in the *LIVE*. When (backward) control flow edges are joined, the *LIVE* sets of different edges will be merged with a union operation. The data-flow equations are formally summarized by Equation 4.1.

$$\begin{cases} LIVEOUT[i] = \bigcup\limits_{s \in succ[i]} LIVEIN[s] \\ LIVEIN[i] = \begin{cases} LIVEOUT[i].\text{prefixsub}(LHS[i], RHS[i]) & \text{if } i \text{ is a ref copy} \\ USE[i] \cup (LIVEOUT[i] - DEF[i]) & \text{otherwise} \end{cases} \end{cases} \tag{4.1}$$

where *LIVEIN*[i] and *LIVEOUT*[i] define the sets of live access paths before and after statement i, respectively, and $succ[i]$ consists of the successor statements of i. By solving the above data-flow equations iteratively and inter-procedurally, the *LIVEIN* sets will converge to a fix-point. Finally, the analysis outputs the *LIVEIN* at the entry as the *LIVE* for this callback (i.e., *LIVEIN* at line 4).

A couple of details of the above static analysis are worth mentioning. First, it does not analyze any methods that are not rooted in the activity instance itself (i.e., `this`), such as the method `getTime()` at line 7 or any constructor like `new A()`, because they are out of the scope of the activity instance state. Second, it is possible that an access path gets killed via some alias that we are not tracking, in which case a must-alias analysis could be incorporated if imprecision is observed to be excessive; we did not observe this in our experiments.

In practice, there could be complexities that prevent the analysis from being fully field-sensitive, in which cases we may have to conservatively save and restore their ancestor objects. Next, we discuss some of these cases. First, when we discover a repeating access path for a recursive type, we bound the access path and retain the prefix of the access path before the cycle, to ensure the entire data structure is retained for soundness. Second, accesses to collections (such as `List` and `Set`) or arrays are treated index-insensitively. Finally, for Android APIs whose implementations are not part of the application package (APK file), we conservatively mark the base access path and the parameters with "USE". For example, at line 12 in Code 4.1, the analysis marks `this.tView` and `this.t` with "USE". For third-party APIs, the analysis, by default, enters into their implementations to reason about the liveness as long as they are accessible as Java bytecode in the APK file.

As discussed earlier, we only need to perform the entry-liveness analysis on the interaction and post-interaction callbacks. After that, the analysis results of these individual callbacks are aggregated with a union: $LIVE = \bigcup LIVE_i$, where $LIVE_i$ is the result of entry-liveness analysis on the interaction or post-interaction callback i. In fact, our analysis can directly produce the aggregated $LIVE$ by creating a pseudo-callback `root()` that calls all relevant callbacks one by one.

4.3.3 May-Modify Analysis

According to the modification property (mentioned at the beginning of Section 4.3), we only need to save and restore the access paths that may be modified by some UI or system callbacks. As discussed in the callback modeling, UI and system callbacks can only be invoked during the interaction phase. Therefore, we need to perform a *may-modify analysis* on the interaction callbacks.

```
class FooActivity extends Activity{
  ...
  void onClick() {
    if(mCheckBox.isSet() == true) { //MOD = {this.pView, this.u.name, this.p}
      this.u.name = mEditText.getText(); //MOD = {this.pView, this.u.name,
          this.p}
      this.p = this.u; //MOD = {this.pView, this.p}
      this.pView.setText(this.p.name); //MOD = {this.pView}
    }
    //MOD = { }
  }
}
```

Code 4.2: Example of may-modify analysis.

Code 4.2 shows an intuitive example of the may-modify analysis on callback `onClick()`. We perform may-modify analysis as a backward analysis. Given a callback, may-modify analysis starts from the exit of the callback with an empty *MOD* set (see line 9 in Code 4.2). Going backwards, for each statement that modifies a field, the analysis adds the statement's *DEF* access path to the current *MOD*. For a local variable write `x = RHS`, we substitute the RHS access path for all occurrences of `x` in *MOD*. When (backward) control flow edges are joined, the *MOD* sets of different edges will be merged with a union operation. Data-flow Equation 4.2 formally captures this analysis.

$$\begin{cases} MODOUT[i] = \bigcup_{s \in succ[i]} MODIN[s] \\ MODIN[i] = \begin{cases} MODOUT[i].\text{prefixsub}(DEF[i], RHS[i]) & \text{if } DEF[i] \text{ is a local var} \\ MODOUT[i] \cup DEF[i] & \text{if } DEF[i] \text{ is a field} \end{cases} \end{cases} \tag{4.2}$$

Here *MODIN*[i] and *MODOUT*[i] define the sets of may-modify access paths before and after statement i. By solving the above data-flow equations inter-procedurally, the analysis finally outputs the *MODIN* at the entry of this callback (i.e., the *MODIN* set at line 4).

Similar treatments for the complexities discussed in entry-liveness analysis are also applied here. In particular, an assignment to a local access path with value unreachable from the activity removes all the descendants of this local access path in *MOD*, as it is not part of the activity's instance state. Additionally, if at any point a *MOD* set contains access paths a_i and a_j such that a_i is a prefix of a_j, a_j can be removed, since the analysis treats the presence of a_i in *MOD* as meaning that *all* state reachable from a_i (including a_j) may be modified. Assuming the may-modify analysis result for callback i is MOD_i, then the aggregated analysis result would be $MOD = \bigcup MOD_i$.

So far, we have introduced the internal state identification in Java `Activity` class. Next, we will discuss the external state identification regarding the GUI elements. Note that while GUI element references in the `Activity` class are part of the internal state, the actual objects are usually declared in XML files. As we will elaborate next, the above analyses cannot cover the all GUI elements.

4.3.4 UI Property Analysis

As mentioned in Chapter 2, besides a Java class, an activity also contains layout files (in XML) for declaring and organizing the GUI elements. Android compiles the layout files at runtime and provides APIs (typically `findViewById()`) for accessing their GUI elements. Once the GUI elements are referenced in the `Activity` class, our previous analyses (entry-liveness and may-modify analyses) can determine whether their references should be preserved. But they are insufficient to cover all the GUI elements that should be preserved. First, unlike a Java object which becomes useless if its last reference is "killed", a GUI element is still useful ("read" by users) even when all its references in the `Activity` class are "killed." Second, for GUI elements that are never referred in the `Activity` class, their properties may still be modified by the user directly. Essentially, GUI elements can be "read" and "modified" via a non-programmatic channel – direct user interaction. For the above reasons, we developed a separate analysis, called *UI property analysis*, to find the properties of GUI elements that are necessary to preserve during activity restarts and app relaunches. Note that, besides those statically declared in the XML layout files, some GUI elements may also be declared in the `Activity` class, referred to as *dynamic GUI elements*. The UI property analysis covers both kinds of GUI elements.

The output of UI property analysis is called the *external state*. For the *liveness* aspect of external state, we assume all the GUI elements are *live* as they are visually "read" by users. In the following, we focus on the *modification* aspect of external state. First, like the internal state, only modifications to the external state during the interaction phase should be analyzed. However, unlike the internal state, the external state can be either directly modified by the user or programmatically updated by UI and system callbacks through references. Thus, we separate the "*MOD*" of the external state into two parts: (i) *EUI* – the (user) editable properties of all the declared GUI elements; and (ii) *MUI* – the GUI properties (including the non-editable ones) that may be modified by UI and system callbacks. For *EUI*, we first list the editable properties for each type of GUI element in Android.[5] Given this list, the analysis scans the layout files and the `Activity` class to identify all the statically and dynamically declared GUI elements, and then outputs their editable properties. For *MUI*, the analysis first searches the UI and system callbacks transitively for GUI element access APIs that modify GUI properties, and then marks such properties of all declared GUI elements of the same type as *MUI*. For example, a call to `tempTextView.setText()` puts the property `text` of all declared `TextView` elements to *MUI*.

In summary, similar to the internal state analysis, UI property analysis finds the corresponding "*LIVE*" and "*MOD*" for the external state, where "*LIVE*" includes all properties of declared GUI elements while "*MOD*" is the union between *EUI* and *MUI*.

[5]Note that this is manual effort once for the Android library.

4.3.5 Data Integration

Finally, we can integrate the results from all the prior static analyses to compute the overall necessary instance state to preserve, which can be summarized by the following equations.

$$\begin{cases} \mathit{NISTATE}_{in} = \mathit{MOD} \cap_{alias,field} \mathit{LIVE} \\ \mathit{NISTATE}_{ex} = \mathit{EUI} \cup \mathit{MUI} \\ \mathit{NISTATE} = \mathit{NISTATE}_{in} \cup \mathit{NISTATE}_{ex} \\ [\mathit{NISTATE}, \mathit{aliases}] = \mathrm{aliasing}(\mathit{NISTATE}, \mathit{LIVE}) \end{cases} \tag{4.3}$$

where $\mathit{NISTATE}_{in}$ and $\mathit{NISTATE}_{ex}$ represent the internal and external necessary instance state, respectively. Operation aliasing($\mathit{NISTATE}$, LIVE) finds aliasing relations of the access paths between $\mathit{NISTATE}$ and LIVE. We elaborate these equations next.

Complexities in Data Integration. For the external state, the union operation $\mathit{EUI} \cup \mathit{MUI}$ simply combines the two sets together. However, for the internal state, the intersection $\mathit{MOD} \cap \mathit{LIVE}$ needs to be both *alias-aware* and *field-sensitive* to be *safe* and *precise*.

- First, consider two access paths `this.a.b` and `this.b`, where `this.a.b` $\in \mathit{MOD}$ but `this.a.b` $\notin \mathit{LIVE}$ and `this.b` $\in \mathit{LIVE}$ but `this.b` $\notin \mathit{MOD}$. A conventional intersection $\mathit{MOD} \cap \mathit{LIVE}$ will exclude both `this.a.b` and `this.b`. However, this may be unsafe as `this.a.b` and `this.b` might be aliases, in which case both of them should be included in $\mathit{MOD} \cap \mathit{LIVE}$. To address this hazard, we perform *may-alias analysis* over the two sets MOD and LIVE, and make the intersection *alias-aware*. However, it is well-known [145] that may-alias analysis may suffer from over-approximation. As a result, the intersection may be unnecessarily large. We will address the issues related to aliases in *NISTATE* in detail shortly.

- Second, the intersection $MOD \cap LIVE$ should also be *field-sensitive* to preserve precision. For example, if `this.a` $\in MOD$ and `this.a.b` $\in LIVE$ (or vice versa), the intersection should include `this.a.b`, but not `this.a`. In fact, `this.a` is safe but `this.a.b` is optimal, reducing the amount of data saved and restored (as shown later, our runtime can save and restore partial objects). To achieve field sensitivity, the intersection requires checking for prefixes in the access paths.

Addressing Aliases. Besides the access paths in *NISTATE*, some aliasing relations, in particular, those related to reference comparisons (e.g., `if (this.a == this.b)`), may also need to be preserved to ensure correctness. In fact, not all references involved in reference comparisons need to be preserved. Considering two references in a comparison, say `this.a` and `this.b`, their aliasing relation needs to be preserved only if one of the references is in *NISTATE* and the other is in *LIVE*, because (i) both references need to be live so that the comparison will be useful; (ii) at least one of the references may be modified so that the comparison is non-trivial (i.e., the boolean value may be changed). The operation aliasing(*NISTATE*, *LIVE*) shown in Equation 4.3 finds the aliasing relations satisfying the conditions. Besides correctness, an additional benefit of finding such aliasing relations is to avoid duplicate saving and restoring. Consider two access paths `this.a` and `this.b` in the *NISTATE*; if they are known as aliases, only one of them needs to be saved and restored, while the other can be simply redirected to the restored one. However, as aliasing analysis might be imprecise, we cannot completely rely on it for correctness. To remedy the precision limitation, we will leverage the runtime module to dynamically check (ground) the aliasing relations.

In summary, with the above integration, the static analysis module finally produces the state to preserve *NISTATE* and the associated potential aliasing relations *aliases*.

4.4 Runtime Module

LiveDroid's runtime module carries out two tasks: (i) grounding potential aliases and (ii) managing the state saving/restoring, including data serialization and deserialization.

4.4.1 Alias Grounding

The task of alias grounding is to verify the statically identified aliases among access paths as in $[NISTATE, aliases]$ in Equation 4.3 are actual or not at the time the activity is about to get destroyed. As discussed earlier, the main reason for alias grounding is for correctness – precisely preserving the exact aliasing relations is critical for preserving the values of reference comparisons. Moreover, finding out the actual aliasing relations can avoid duplicated saving of objects.

To implement the alias grounding, LiveDroid saves and restores the actual alias relations as boolean values, along with access paths in *NISTATE*. Furthermore, LiveDroid inserts condition checks right before saving references in each alias class. If some of these references are actual aliases, only one copy of the object they point to is saved. Later, to restore the activity instance state, LiveDroid restores the objects and references based on the recovered alias relations, such that aliased references remain aliases. An example will be provided shortly in the next section.

4.4.2 State Saving and Recovering

As mentioned in Chapter 2, there are two basic approaches for preserving data: (i) saving/restoring the instance state, and (ii) using the `ViewModel`. We choose the first approach, because `ViewModel` requires significant refactoring to adopt and it cannot survive system-initiated killing. Code 4.3 shows an implementation of the first approach by overwriting the saving and restoring callbacks.

Saving/Restoring Internal State. For primitive variables (e.g., `this.x` of type `int`), saving and restoring is intuitive, as shown at lines 3 and 11. A unique key is used for saving and retrieving the variable in the `Bundle` object with APIs matched with its primitive type. For non-primitive types, if the references point to GUI elements (like `this.dView`), we handle them together with the external state; otherwise, we serialize their corresponding objects to strings before saving (line 4) and deserialize the strings back to objects after restoring (lines 12–13). For serialization and deserialization, we leverage the widely used Gson [98] library to convert Java objects to JSON strings and vice versa. The example also shows the alias grounding, which checks potential aliases at runtime and only save and restore one copy of the corresponding object (lines 5–8 and 15–19). For access paths with levels deeper than the `Activity` fields (e.g, `this.a.b`), there are a couple of complexities for saving and restoring, which we discuss and address next.

```
class FooActivity extends Activity{
  void onSaveInstanceState(Bundle state) { ...
    state.putInt("int_x", this.x); //primitive
    state.putString("obj_b", gson.toJson(this.b)); //object
    if(this.a.b == this.b)
      state.putBoolean("a_b=b", true); //save the alias relation
    else //field
      state.putString("obj_a_b", gson.toJson(this.a.b));
  }
  void onRestoreInstanceState(Bundle savedState) { ...
    this.x = savedState.getInt("int_x");
    String str = savedState.getString("obj_b");
    this.b = gson.fromJson(str, B.class);
    is_alias = savedState.getBoolean("a_b=b", false);
    if(is_alias)
      this.a.b = this.b;
    else {
      String str = savedState.getString("obj_a_b");
      this.a.b = gson.fromJson(str, B.class);
    }
  }
}
```

Code 4.3: Saving and restoring internal state.

- ***Handling Private Fields.*** Private fields are not accessible outside their class. For example, assume `this.a.b` is in *NISTATE* but `b` is a private field of `a`. In this case, we cannot access the private field as in the serialization call `gson.toJson(this.a.b)`. Instead, we have to call a getter method like `getB()` that returns the private field `b`, as in `gson.toJson(this.a.getB())`. Similarly, to restore a private field, we need to call its setter method (e.g., `this.a.setB()`). For private fields without getter and setter methods, LiveDroid offers automatic generations of such methods under the direction of developers. Alternatively, we can move up along the access path (e.g., `this.a.b.c` $\rightarrow$ `this.a.b`), until we reach a publicly accessible field or a field that developers are comfortable to add setter/getter methods. In the worst case, the `Activity` field (e.g., `this.a.b.c` $\rightarrow$ `this.a`) can be saved and restored. In general, this option compromises the precision of the app state, thus may increase the cost of state saving/restoring.

- ***Handling Subfields.*** During the reconstruction of access paths, the parent objects need to be constructed before the construction of their child objects. For example, before `this.a.b` is restored, `this.a` must be constructed first; otherwise a null pointer exception will be thrown when `this.a.b` is accessed (e.g., at lines 16 and 19 in Code 4.3). There are two scenarios for the parent object construction. If the default constructor of the parent (e.g., `A()`) is available, we simply invoke it before constructing the child object; Otherwise, if the parent object has an overridden constructor (like `A(B b)`), then we can generate a "default" constructor that carries no parameters (i.e., `A()`) and use it for constructing the parent object. After the construction of the parent object, we can construct the child object and assign it to the corresponding field of the parent object (like `this.a.b = this.b`). If the corresponding field is private, then we can solve it using the solution just mentioned in the prior paragraph – either generating a setter method for the private field or moving up in the access path to save an ancestor object.

Note that saving and restoring a subset of the app state (i.e., the *NISTATE*) can potentially violate *object invariants* [117]. Recall the two groups of access paths that LiveDroid does not save and restore: (i) access paths that remain unchanged during interaction and (ii) access paths that will be redefined before they get used. The first category clearly will not violate object invariants. For the second category, failing to restore these access paths may break some object invariants. However, this violation is temporary and inconsequential because the analysis ensures that the values of these access paths will not be read until they are redefined. After the redefinition, the object invariants are re-established. Hence app semantics are unchanged.

Saving/Restoring External State. For the external state ($EUI \cup MUI$), saving and restoring fall into two cases: First, for editable properties of built-in GUI elements (e.g., `text` of `EditText`), Android offers automatic saving and restoring as long as their instances are declared with IDs. For those without IDs, we assign IDs in the places where the GUI elements are declared. Second, for editable properties of customized GUI elements (defined by developers) and non-editable properties that may be modified by UI and system callbacks (*MUI*), we preserve them using the state saving/restoring callbacks, as shown in Code 4.4. Note that here we distinguish between the static and dynamic GUI elements. For static GUI elements, they are recreated automatically by Android, thus we only need to retrieve them (via `findViewById()` at line 10) and restore their properties (line 11). Furthermore, if there

```java
class FooActivity extends Activity{
  void onSaveInstanceState(Bundle state) { ...
    TextView view = findViewById(R.id.text_time);
    state.putString("text_time", view.getText()); //property of static view
    state.putBoolean("tView_ref", this.tView.getViewId() == R.id.text_time);
         //ref
    state.putString("dView", this.dView.getText()); //property of dynamic
         view
    state.putId("dView_parent", getParentId(this.dView)); //parent GUI of
         dyn. view
  }
  void onRestoreInstanceState(Bundle savedState) { ...
    TextView view = findViewById(R.id.text_time);
    view.setText(savedState.getString("text_time")); //restore property
    if(savedState.getBoolean("tView_ref", false))
      this.tView = view; //redirect GUI reference to new GUI instance
    if(this.dView == NULL) { //if not created during pre-interaction
      this.dView = new TextView(this);
      View parent = findViewById(savedState.getInt("dView_parent"));
      parent.add(this.dView); //attach the dynamic GUI to its parent GUI
    }
    this.dView.setText(savedState.getString("dView")); //restore property
  }
}
```

Code 4.4: Saving and restoring external state.

are corresponding references found in $NISTATE_{in}$, we redirect those references to the newly created GUI elements. For dynamic GUI elements, we need to recreate them first (line 15) and attach them to their parent GUI elements (lines 16-17), then recover their properties (line 19).

The aforementioned external state saving/restoring strategy assumes that the GUI elements and their appearances should remain the same after restarting or relaunching. In certain cases of activity restarting, however, developers may want to change the GUI elements and/or their appearances after restarting (e.g., changing the appearances of some GUI elements after the phone rotation). For such cases, developers can step in and bypass the saving and restoring of relevant parts of the state to avoid overwriting their customized setups of GUI elements and properties, which are usually specified in a different layout file.

Handling Asynchronous Tasks. An activity instance may offload some blocking tasks (e.g., downloading a file) asynchronously to background threads to keep the UI thread responsive. Android offers several ways to achieve this; commonly used strategies include `AsyncTask`[52], `Service`[66], and `IntentService`[63]. First, as these components are separated from `Activity`, they can continue executing on the background threads during the activity restarting. However, upon app relaunching, their execution will be terminated along with the activities. Unfortunately, unlike `Activity`, there are no dedicated saving and restoring mechanisms offered by Android for these components to preserve their states before they get destroyed. On the other hand, this design aligns with the nature of the asynchronous tasks – they are temporary and can be relaunched as needed. So, instead of preserving their states upon destroying/termination, we may preserve their initial states – their “inputs”, so that they can be relaunched. As long as these asynchronous tasks do not depend on the activity instance state, this handling will not affect the correctness. Actually, such handling has been provided by Android for `IntentService`. The `Bundle` that serves as the input to an `IntentService` is automatically saved, and then it is reused when the `IntentService` is relaunched (following the app relaunching). For `Service`, similar handling can be easily enabled via a flag named `START_REDELIVER_INTENT`. For `AsyncTask`, Android does not offer a similar service. In order to preserve the input parameters of an `AsyncTask`, we need to locate the callsite where the `AsyncTask` was launched, save and restore its parameters along with the activity instance state. An alternative solution is to refactor the `AsyncTask` to an `IntentService`. In fact, prior work [124] has shown that `AsyncTask` is the source of many runtime issues and the more recent component `IntentService` is preferred.

4.5 Implementation

As some implementation details have already been discussed, in this section we focus more on the tools that realize the static analyses and runtime module; including a *static analyzer*, an *Android Studio plugin*, and an *APK patching tool*. The latter two are alternative ways to help developers generate state-saving/restoring routines. Together, they constitute LiveDroid.

Static Analyzer. The static analyzer, namely LiveDroid-analyzer, is implemented using several program analysis libraries built upon the Soot [144] framework, including Heros [82], Spark [118], and FlowDroid [76]. The Soot [144] framework provides an easy-to-manipulate intermediate representation (Jimple) for analyzing Java programs, and Heros [82] provides a solver for inter-procedural, finite, distributive subset (IFDS) problems in a flow-sensitive and context-sensitive manner. LiveDroid-analyzer takes an app's APK file as input and feeds it into FlowDroid [76] to collect user interaction and post-user interaction callbacks in the form of inter-procedural control-flow graphs (ICFGs). By traversing the ICFGs, LiveDroid-analyzer then identifies the GUI properties that may be modified by callbacks (*MUI*). Next, the ICFGs are passed to Heros where the entry-liveness and may-modify analyses are implemented. The analysis results are then integrated (i.e., $MOD \cap_{alias,field} LIVE$) with the help of an alias analysis module in Soot, called Spark [118]; Spark implements Andersen's points-to analysis. The editable GUI properties (*EUI*) are captured in a manually-constructed list, based on Android APIs. Developers may expand this list with editable properties of their customized GUI elements. Finally, LiveDroid-analyzer outputs the aggregated analysis results into a report file in XML.

Per the IFDS framework [134, 82], the worst-case time complexity for our entry-liveness and may-change analyses (two locally separable problems) is $O(ED)$, where E is the number of edges in the supergraph, and D is domain size, that is the number of access paths. The time complexity for pointer analysis using Spark [118] is cubic in program size for typical inputs.

Android Studio Plugin. We developed a plugin based on Android Studio 3 – LiveDroid-plugin, which can generate code that realizes the runtime module either for an activity or the whole app. To do so, LiveDroid-plugin first takes the report from static analyzer as an input and extracts the static analysis results. Then, when directed by developers, it generates constructors for classes missing default constructors and getter/setter methods for private fields that need to be accessed. Finally, the plugin inserts the saving and restoring code into callbacks `onSaveInstanceState()` and `onRestoreInstanceState()` for each access path together with the alias grounding code for each group of potentially aliased references specified in the static analysis report. The plugin can help developers refactor their code based on their needs.

APK Patching Tool. As an alternative solution, we also developed a patching tool – LiveDroid-patch, which can directly insert code into a compiled app (APK), without accessing the source code. The tool uses the static analysis report as the plugin does, based on which it injects the data saving and restoring code into the APK file. To achieve this, LiveDroid-patch leverages Soot for reverse engineering, code insertion, and recompilation. After that, `Zipalign` and `apksigner` are used to align and sign the final APK. Note that LiveDroid-patch avoids the complexities of accessing private fields and constructing parent objects (see Section 4.4), as it directly modifies the binary.

Limitations. Although LiveDroid aims for an accurate solution with a combination of static analysis and runtime modules, the static analysis inherits limitations from other static analysis tools. For example, FlowDroid [76] does not support lambda-style event declarations in Java 8 and native method modeling. It also inherits limitations on reflective calls, which are resolved only if their arguments are string constants. The LiveDroid analysis and plug-in can only process Java code at the moment, as Soot does not fully support the `invokedynamic` bytecode [93] which affects apps written in Kotlin (or Java code using lambdas and method refs). Similar limitations apply to the plug-in, which is written for Java only, and cannot process `NativeActivity` [57] or apps written in Kotlin.

4.6 Evaluation

This section evaluates LiveDroid on real-world Android apps to demonstrate its applicability, effectiveness in identifying the *NISTATE*, its costs and benefits for generating state-saving and restoring routines, as well as some of its limitations.

4.6.1 Methodology

To evaluate LiveDroid, we crawled 1,033 packages from F-Droid [89] app store and retained 966 apps (denoted as *GroupL*); apps without activities, or for which FlowDroid failed to build an ICFG, were removed. We selected an additional set of 36 apps (denoted as *GroupS*), as they had necessary instance states and at least 20 Stars on GitHub [94] or 5K downloads on GooglePlay [99]. As shown in Table 4.4, they include some highly influential open-source projects, such as K-9 mail, Free RDP, and LeafPic. Together, *GroupL* and *GroupS* are composed of 4,808 and 231 activities, respectively. We use *GroupL* for evaluating the applicability of the static analyzer in general and *GroupS* for a focused study, including collecting the statistics of the app state *NISTATE* and analyzing the costs and benefits of state saving and restoring. For each Android project, we applied LiveDroid to each activity registered in the `AndroidManifest.xml`.

To examine the actual app behaviors, we use a Nexus 5X smartphone running Android version 8.1. The programming environment is Android Studio 3.4. Experiments on *GroupL* were conducted on a PC with a 3.5 GHz Intel Xeon processor and 16 GB RAM, while results for *GroupS* were collected on a MacBook Pro with a 2.0 GHz Intel Core i5 processor and 8 GB RAM.

4.6.2 Static Analysis

We evaluated the applicability of the static analyzer on *GroupL* and the static analysis results on *GroupS* in detail, including performance, app state statistics, and correctness. In summary, our evaluation results show that the proposed static analysis is generally applicable to various real-world apps, effective in identifying necessary instance state – yielding significantly fewer access paths compared to previous work [142] but being more systematic than manual state identification. In the following, we first present the applicability results of the static analyzer, then discuss the detailed static analysis results on *GroupS*.

Applicability. Table 5.2 summarizes the static analysis results on *GroupL*. At the activity level, among 4,808 activities, 1,896 activities contain non-empty state *NISTATE*, including 1,630 activities (33.9%) with non-empty external state and 512 activities (10.6%) with non-empty internal state. At the app level, among 966 apps, 452 (46.8%) contain at least one activity with non-empty external state and 322 (33.3%) contain at least one activity with non-empty internal state. Note that the above results are from static analysis, rather than the ground truth. Later in this section, we will report the number of false positives in the static analysis results when we study *GroupS*. While we did not observe any failures during the above analysis, there are a couple of situations where the static analyzer may fail, including `NativeActivity` written in C/C++ and activities implemented in Kotlin (Soot does not fully support `invokeDynamics` [93]).

Time Cost. We measured the time required for performing the static analysis on *GroupS*. The results are reported in Table 4.4 under Column "Time". For most apps (30/36), the

Table 4.2: Activities and Apps (*GroupL*) with Non-Empty External/Internal States.

Necessary instance state (*NISTATE*)	#activities	#apps
Non-empty external state ($NISTATE_{ex} \neq \emptyset$)	1630 (33.9%)	452 (46.8%)
Non-empty internal state ($NISTATE_{in} \neq \emptyset$)	512(10.6%)	322 (33.3%)

analysis finishes within 1 minute and often within 10 seconds. For app#28, the analysis took much longer, nearly 30 minutes. After examining its source code, we found the app uses multiple external libraries (e.g., Dropbox, JodaTime and Apache Jackrabbit), which greatly increases the analysis workload. This problem can be mitigated with the help of developers by specifying the source code packages that the analyzer may skip, a functionality we plan to add later.

State Statistics. The detailed analysis results on *GroupS* are reported in Table 4.4. First, Column EX reports the size of external state $NISTATE_{ex}$ in terms of the number of properties. We found that 21 out of 36 apps have necessary GUI properties that must be preserved. Among them, app#27 has the most – 35 GUI properties, due to its richer and more interactive user interface, which take more inputs from the user. The next column, UIC, reports the number of GUI elements with at least one necessary access path. Comparing this column with the prior one, we can find that most GUI elements have just one necessary access path. The next two columns, MOD and LIVE, show the sizes of *MOD* and *LIVE*, respectively. In general, there are more access paths in *LIVE* than *MOD*, indicating that some access paths remain unchanged through the app lifecycle (e.g., "constants"). As mentioned earlier, the intersection between *MOD* and *LIVE* defines the internal state $NISTATE_{in}$, whose size is reported in Column IN. As the results show, IN is consistently less than *MOD*

Table 4.3: Details of selected apps in *GroupS* to apply LiveDroid-Analyzer.

Stars: #stars on GitHub, Popu.: #downloads on Google Play, ACT: #activities,

MUT: #access paths of all mutable activity fields (up to 3rd level) – state in [142]

#	Package	Stars	Popu.	ACT	MUT
1	au.com.wallaceit.reddinator	30	50K+	18	1,529
2	com.alaskalinuxuser.hourglass	5	5K+	2	25
3	com.dozingcatsoftware.asciicam	105	100K+	5	102
4	com.freerdp.afreerdp	4,297	100K+	30	254
5	com.fsck.k9	4,659	5M+	29	3,275
6	com.github.axet.binauralbeats	16	50K+	23	549
7	com.github.xloem.qrstream	29	-	4	25
8	com.ihunda.android.binauralbeat	135	1M+	1	225
9	com.kiminonawa.mydiary	1,402	-	14	2,542
10	com.llamacorp.equate	45	10K+	13	201
11	com.namelessdev.mpdroid	557	100K+	26	172
12	com.ringdroid	692	-	3	142
13	com.sagar.screenshift2	47	1M+	36	297
14	com.tastycactus.timesheet	42	-	6	67
15	de.baumann.browser	387	10K+	12	319
16	de.onyxbits.listmyapps	57	100K+	5	69
17	de.schildbach.wallet	1,818	1M+	13	415
18	de.smasi.tickmate	78	1K+	10	173
19	jackpal.androidterm	2,318	10M+	21	185
20	jp.sblo.pandora.aGrep	18	10K+	21	65
21	moe.minori.pgpclipper	18	-	5	54
22	net.kervala.comicsreader	1	100K+	11	166
23	nl.asymmetrics.droidshows	53	-	6	283
24	org.billthefarmer.diary	98	-	3	48
25	org.billthefarmer.tuner	91	-	3	171
26	org.disrupted.rumble	138		23	553
27	org.glucosio.android	324	-	17	621
28	org.gnucash.android	987	100K+	22	1,169
29	org.horaapps.leafpic	2,948	-	11	3,038
30	org.openintents.notepad	38	50K+	9	127
31	org.secuso.privacyfriendlynotes	46	5K+	11	145
32	org.secuso.privacyfriendlytapemeasure	8	5K+	11	644
33	org.yaxim.androidclient	91	100K+	10	205
34	ru.henridellal.emerald	42	10K+	8	108
35	se.bitcraze.crazyfliecontrol2	85	10K+	12	713
36	tellh.com.gitclub	620	-	15	1,102
Total				469	19,778

Table 4.4: Detailed static analysis results from applying LiveDroid-Analyzer on *GroupS*.
EX: #access paths in the external state, UIC: #necessary GUI elements, MOD: #access paths in set *MOD*,

LIVE: #access paths in set *LIVE*, IN: #access paths in the internal state,

S: #access paths in set *NISTATE*, Alias: #alias groups (#total aliases), Time: analysis time (s)

#	Package	EX	UIC	MOD	LIVE	IN	S	Alias	Time
1	au.com.wallaceit.reddinator	32	23	42	99	7	39	0	21
2	com.alaskalinuxuser.hourglass	0	0	13	15	6	6	0	2
3	com.dozingcatsoftware.asciicam	0	0	7	31	5	5	1(2)	3
4	com.freerdp.afreerdp	0	0	12	55	10	10	1(2)	14
5	com.fsck.k9	0	0	59	241	25	25	1(2)	62
6	com.github.axet.binauralbeats	1	2	90	252	49	50	0	40
7	com.github.xloem.qrstream	2	2	12	28	10	12	0	54
8	com.ihunda.android.binauralbeat	0	0	0	11	0	0	1(2)	3
9	com.kiminonawa.mydiary	0	0	8	31	8	8	0	24
10	com.llamacorp.equate	0	0	27	86	25	25	0	7
11	com.namelessdev.mpdroid	5	5	140	172	58	63	1(2)	42
12	com.ringdroid	0	0	18	30	10	10	0	7
13	com.sagar.screenshift2	10	10	12	45	0	10	0	11
14	com.tastycactus.timesheet	19	19	10	52	8	27	0	2
15	de.baumann.browser	17	12	31	61	24	41	0	11
16	de.onyxbits.listmyapps	10	10	5	15	4	14	0	5
17	de.schildbach.wallet	0	0	24	61	17	17	0	507
18	de.smasi.tickmate	13	13	36	22	10	23	0	6
19	jackpal.androidterm	0	0	53	98	34	34	0	61
20	jp.sblo.pandora.aGrep	0	0	2	30	1	1	0	1
21	moe.minori.pgpclipper	0	0	15	33	12	12	0	5
22	net.kervala.comicsreader	2	2	41	97	29	31	1(2)	9
23	nl.asymmetrics.droidshows	30	19	7	43	5	35	0	7
24	org.billthefarmer.diary	5	4	4	16	2	7	1(2)	4
25	org.billthefarmer.tuner	0	0	6	6	6	6	0	16
26	org.disrupted.rumble	13	10	20	48	13	26	0	24
27	org.glucosio.android	35	35	1	7	0	35	0	116
28	org.gnucash.android	18	10	3	29	2	20	1(2)	1,779
29	org.horaapps.leafpic	0	0	20	57	13	13	0	47
30	org.openintents.notepad	2	2	20	38	12	14	1(2)	6
31	org.secuso.privacyfriendlynotes	22	15	21	47	16	38	0	10
32	org.secuso.privacyfriendlytapemeasure	5	5	21	43	15	20	2(4)	6
33	org.yaxim.androidclient	0	0	10	25	8	8	0	68
34	ru.henridellal.emerald	7	7	17	27	6	13	0	5
35	se.bitcraze.crazyfliecontrol2	7	7	55	189	34	41	2(4)	22
36	tellh.com.gitclub	0	0	10	84	7	7	0	49
Total		255	212	872	2,224	491	746	13	-

and *LIVE* among all tested apps, except for app#8, app#9, and app#25, in which cases IN = MOD. Adding the internal state size IN and external state size EX together, the overall app state size is shown in Column S (i.e., S = IN + EX). Finally, as discussed in Section 4.3, some access paths in the *NISTATE* may be aliases. Column ALIAS shows the number of alias groups and the number of aliases in total, which (1) confirms the necessity of aliasing analysis, and (2) shows that aliases do not occur often or in large groups in the *NISTATE*.

State Comparison with Prior Work. It is important to note that internal necessary instance state (Column IN) is significantly smaller than the number of mutable activity fields in the app (shown in Column MUT) – the app state considered by recent work [142]. On average, IN *is only 1.5%* of MUT. The reduction mainly comes from a more rigorous definition of the necessary instance state based on the *liveness* and *modification* properties, as well as field-sensitive analysis results (i.e., access paths). Moreover, the prior work [142] does not cover the external state which also represents a large portion of the total necessary instance state (see Column EX).

Revealed State Issues. We manually compared the state identified by the static analyzer (Column S in Table 4.5) with the state actually saved by developers in the original app code (Column S_s in Table 4.5). Interestingly, we found that *a large number of necessary access paths are not saved and restored*, as reported in Column S_u (i.e., S - S_s). In total, there were 231/393 identified access paths (including GUI properties and activity fields) that were not saved and restored in the original apps, which may lead to various state issues during the user interaction. To confirm this, we manually tested activities of each app in *GroupS* based on the identified unsaved access paths S_u to verify if they cause any state issues. The

Table 4.5: New Issues discovered by applying LIVEDROID-ANALYZER on *GroupS*.
S: size of *NISTATE*, ISSUE$_d$:#issues detected, S$_u$: #access paths of unsaved *NISTATE*,

S$_s$: #access paths of saved *NISTATE*, FP% : false positive ratio

Package	S	ISSUE$_d$	S$_s$	S$_u$	FP%
com.alaskalinuxuser.hourglass	6	1	0	6	0
com.fsck.k9	25	2	23	2	0
com.github.xloem.qrstream	12	1	0	10	16.7
com.kiminonawa.mydiary	8	3	0	8	0
com.ringdroid	10	1	0	7	30.0
com.tastycactus.timesheet	27	4	1	24	7.4
de.baumann.browser	41	3	14	27	0
de.smasi.tickmate	23	2	4	13	26.1
moe.minori.pgpclipper	12	3	2	9	8.3
nl.asymmetrics.droidshows	35	2	32	2	2.9
org.billthefarmer.diary	7	4	3	4	0
org.billthefarmer.tuner	6	1	0	6	0
org.disrupted.rumble	26	2	13	4	34.6
org.glucosio.android	35	3	31	4	0
org.gnucash.android	20	1	1	17	10.0
org.horaapps.leafpic	13	1	3	10	0
org.openintents.notepad	14	3	1	10	21.4
org.secuso.privacyfriendlynotes	38	6	20	18	0
org.secuso.privacyfriendlytapemeasure	20	1	5	15	0
org.yaxim.androidclient	8	1	3	3	25.0
tellh.com.gitclub	7	1	6	1	0
Total	393	46	162	200	7.9

testing results confirm that 200 out of 230 unsaved access paths *do trigger state issues.* The number of issues from the user's perspective is reported under Column ISSUE$_d$. Note that a state issue often involves multiple (necessary) access paths in the app state. Most issues are manifested as the loss of some user interaction state. Table 4.6 reports some examples of the newly-revealed state issues exposed by our approach. As we will show later, all these new state issues can be fixed by the runtime module with generated state-saving/restoring routines.

```
// activity_bootloader.xml
<RelativeLayout xmlns:android="http://schemas.android.com/..." ...>
  <TextView android:id="@+id/bootloader_title" ... /> //false positive
  ....
  <TextView android:id="@+id/bootloader_statusLine" ... /> //true
      positive
  ...
</RelativeLayout>
// BootloaderActivity.java
public class BootloaderActivity extends Activity {
  private TextView mConsoleTextView; //true positive
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_bootloader);
   this.mConsoleTextView = (TextView)
       findViewById(R.id.bootloader_statusLine);
  }
  protected void onPostExecute(String result) { ... //AsyncTask callback
    appendConsoleError("Firmware file can not be found.");
  }
  public void appendConsoleError(String status) { ...
    this.mConsoleTextView.append("\n" + status);
  }
  public void startFlashProcess(final View view) { //click handler
    this.mConsoleTextView.setText("");
  }
}
```

Code 4.5: Case Study: False Positive and True Positive.

Table 4.6: Example issues revealed by static analysis.

Package	State issues
com.alaskalinuxuser.hourglass	timer state lost and paused
com.fsck.k9	bound email account lost
com.fsck.k9	entered email text lost
de.baumann.browser	webpage reloaded
org.gnucash.android	search results disappear
org.horaapps.leafpic	player position reset to 0

False Positives and False Negatives. Our manual examination also showed that some reported necessary access paths are actually false positives – they do not trigger any actual state issues even when they are unsaved. These access paths are reported in Column FP% of Table 4.5. While false negatives are possible (e.g., due to reflection), we did not find any false negatives in our evaluation. In general, our static analyses are designed to be over-approximate, modulo unhandled language features. Next, we focus our discussions on false positives.

The cost of false positives is extra state saving and restoring; developers are not required to handle false positives. Our examination reveals two main causes of false positives. The first reason is unrealizable execution paths. Like other data-flow analyses, our entry-liveness and may-modify analyses are conservative in the sense that they assume all the control-flow paths are possible. However, depending on the constraints along the paths, many of them may never happen. Similarly, there could be semantic constraints among the GUI elements that restrict the order in which callbacks may be invoked. Our callback modeling does not reason about such constraints. Another cause of false positives is our coarse-grained UI property analysis, which does not distinguish between different GUI elements of the same type. This can be improved by tracking updates to each individual GUI element. The challenge lies in the fact that a GUI element reference in the activity class may refer to different GUI elements (in the layout file) at different times. On the other hand, note that the imprecision of alias analysis does not introduce false positives, because if one reference is true positive, all of its aliases are also true positives – they point to the same object.

Case Study. Code 4.5 presents an example activity from the project `se.bitcraze.crazyfliecontrol2`. First, our entry-liveness and may-modify analyses find that the access path `this.mConsoleTextView` is both live and may be modified (see lines 28 and 32). On the other hand, the UI property analysis finds that the `text` properties of two GUI elements, `bootloader_title` and `bootloader_statusLine`, both belong to the external necessary instance state because some APIs of `TextView` are invoked, `append()` and `setText()`. In total, our static analyses report three positives. However, one of the GUI properties, `bootloader_title.text`, is a false positive as its instance never gets modified in any callback. This false positive comes from the imprecision of the UI property analysis. Furthermore, it is not hard to find that the base objects of the external and internal necessary instance states (`bootloader_statusLine` and `this.mConsoleTextView`) actually refer to the same GUI element (see line 18). This indicates opportunities for improving our static analyses. In fact, if we can infer that a `View` reference always points to the same GUI object, we can save just the GUI object and link the `View` reference to the object during state recovery. We leave the exploration of such improvements for future work.

4.6.3 Generating State-Saving/Restoring Routines

In the following, we first evaluate the applicability of code generation for saving/restoring state using the Android Studio plugin and the APK patching tools, and report the time and space costs of code generation, as well as the runtime costs of state saving and restarting. For the applicability evaluation, we use apps in *GroupS*; For the cost measurements, we randomly select 8 apps from *GroupS* as they involve significant manual efforts. We ensured that the 8 apps include 4 apps with collections (i.e., such as `List` and `Set`) and 4 apps without collections from *GroupS*. The reason we separate these two cases is because the collections, as dynamic data structures, may carry relative larger amount of data, in which case the benefits of reduced state size might be more significant.

Applicability. First, we installed the LiveDroid-Plugin on Android Studio 3.4. Then, for each app in *GroupS*, we loaded its source code into Android Studio and manually went through the code generation process with the installed plugin. We confirmed that the plugin was applied to all the apps successfully. Next, we tested the developed patching tool by first generating APK files for all the apps in *GroupS*. Then, we applied the patching tool to each APK file. Again, we did not observe any issues when using the tool. These results demonstrate the applicability of our developed tools.

Code Generation Costs. The time costs of code generation and patching are both reported in Table 4.7. The "Plugin" column shows the average time cost for applying the code generation, at the activity level, using the plugin. It contains two sub-columns: one for the first time applying and one for the second time. Note that the first-time application incurs more setup costs (e.g., inserting setter/getter methods). On average, the time cost

Table 4.7: Time costs (ms) of code generation.

Package	Plugin		
	1st	2nd	Patching
com.alaskalinuxuser.hourglass	257	184	17,691
com.fsck.k9	672	470	35,676
com.kiminonawa.mydiary	302	220	18,436
com.tastycactus.timesheet	205	135	26,515
de.smasi.tickmate	539	457	19,327
nl.asymmetrics.droidshows	576	351	17,745
org.gnucash.android	729	514	451,231
org.secuso.privacyfriendlynotes	275	231	19,809
Arithmetic Mean	444	320	75,804

Table 4.8: Space costs of LiveDroid-Plugin.

Package	APK (Kilobytes)		Lines of Code (SLoC)	
	before	after	before	after
com.alaskalinuxuser.hourglass	2,028	2,143	18.9K	19.0K
com.fsck.k9	6,562	6,640	214.6K	214.8K
com.kiminonawa.mydiary	16,305	16,306	64.5K	64.6K
com.tastycactus.timesheet	58	206	3.4K	4.1K
de.smasi.tickmate	1,467	1,557	15.8K	16.4K
nl.asymmetrics.droidshows	227	320	9.7K	10.4K
org.gnucash.android	7,882	7,889	109.1K	109.3K
org.secuso.privacyfriendlynotes	2,543	2,633	26.7K	27.2K
Arithmetic Mean	4,634	4,712	57.8K	58.2K

is less than 500ms, hence LiveDroid-Plugin's responsiveness makes it suitable for being used in development environments. The "Patching" column reports the total processing time of each APK file; on average, it takes about 30 seconds.

Besides time costs, code generation also increases the size of source code. Table 4.8 reports the space costs of code generation using the plugin (in terms of #lines of source code) and patching (in terms of APK file size). The increase is relatively small for large apps and more significant for small apps. On (arithmetic) average, there is a 0.7% (58.2K vs. 57.8K) increase in terms of lines of code for using the plugin and a 1.6% (4712KB vs. 4634KB) increase in terms of APK size.

Time Saving with Reduced State Size. We compare the costs of state saving/restoring using LiveDroid with the costs of saving/restoring using all mutable activity fields [142]. To simulate the app restarting scenarios, we turned on the "No background process" option in the test smartphone's Settings – this way the OS will automatically kill an app once it is moved into background and relaunch it once the user switches back to it. Table 4.9 presents the results of time cost for saving and restoring, and speedup gained using LiveDroid, compared to saving and restoring all mutable activity fields. In general, for apps with collections, the speedup can reach over 140X for state saving and over 40X for state restoring. The difference between the speedups for saving and restoring is due to flash memory's asymmetric read/write speeds. For apps without collections, the speedups are relatively smaller, ranging from 1.5-17.5X for state saving and 1.1-6.8X for state restoring, respectively. The exact speedup depends on the reduction in the number of access paths (see columns MUT in Table 4.3 and S in Table 4.4) and the types of the access paths (e.g., a primitive or an object with multiple fields). These results confirm the end-to-end benefits of the proposed state identification techniques (Section 4.3), which shrink the app state substantially, hence substantially reducing the costs of state saving/restoring.

Correctness of Code Generation. To verify if the generated code works correctly or not, we installed all the apps with generated state-saving/restoring routines on a Nexus 5x smartphone and manually examined their behavior. First, we checked the 46 state issues we found based on the static analysis (Section 5.5.2). The results show that all 46 issues were

Table 4.9: Time costs (ms) of saving/restoring before and after LiveDroid-Plugin.

Package	Saving time and speedup			Restoring time and speedup		
	before	after	speedup	before	after	speedup
de.smasi.tickmate	300.3	2.4	123.7X	56.8	1.4	41.0X
nl.asymmetrics.droidshows	283.0	2.0	141.1X	43.8	1.0	43.8X
org.glucosio.android	196.4	5.4	36.3X	29.6	1.4	21.6X
org.secuso.privacyfriendlynotes	71.4	2.3	30.8X	29.9	1.4	21.8X
com.alaskalinuxuser.hourglass	173.6	115.3	1.5X	217.1	205.4	1.1X
com.fsck.k9	156.5	23.8	6.6X	30.3	5.5	5.5X
com.kiminonawa.mydiary	158.2	9.0	17.5X	25.8	3.8	6.8X
org.billthefarmer.diary	244.6	145.7	1.7X	42.3	20.8	2.0X
Geometric Mean	182.9	11.0	16.6X	44.7	4.7	9.5X

successfully fixed, because the issues were due to unsaved necessary access paths, and the generated state-saving/restoring routines ensured these necessary access paths are saved and restored. In addition, we checked if there are any new issues introduced by the code generation; none was observed. This is expected as the data saving itself does not cause any functional side-effect and data restoring occurs after all the initialization operations, ensured by the `onRestoreInstanceState()` callback. In actual development scenarios, the app code may evolve over time. In such cases, developers may re-apply our tools to their applications after subsequent source code changes; this would not lead to issues because our tools only insert saving and restoring code to two callbacks dedicated for state saving and restoring, and new code generations will simply overwrite the previous versions.

4.7 Conclusion

This work targets a major challenge in developing reliable mobile apps – a volatile runtime environment that repeatedly destroys app state through activity and app-level restarts. The solution is an automatic approach that combines callback modeling, static analysis, and runtime data saving/restoring techniques. The callback modeling categorizes different callbacks based on their invocation orders relatively to the user interaction. The static analysis takes both app code (Java class) and GUI interface (layout file) as inputs and identifies critical external and internal app state via novel, necessary instance state, abstraction. The runtime module saves and restores the identified app state, based on the actual aliases. Finally, the evaluation confirms that the developed tool set LiveDroid, including an Android Studio plugin and an APK patcher, is able to find the critical app state from a large space of candidate access paths and substantially boost reliability of apps running in volatile runtime environments through code generation.

Chapter 5

Static User Transaction Graph

5.1 Introduction

As mentioned in earlier chapters, the mobile app market is competitive and mobile apps are highly interactive, thus it is important for app developers to ship apps that provide a great user experience with low user-perceived latency [86] and reliable functionalities.

The primary way to keep UI responsive is to use an asynchronous programming model with multiple threads to handle blocking operations such as handling I/O and data processing [50]. The asynchronous programming model quickly results in complex program flows. For example, a simple data fetching request could trigger multiple asynchronous calls and complex synchronization among threads. In such cases, identifying performance bottlenecks requires tracking of program flows of event handlers defined at main thread (UI thread) and asynchronous threads.

State of The Art. Recent studies [133, 163] highlighted importance and challenges to profile and measure latency in user transaction for mobile apps and systems. AppInsight [133] focuses on Windows Phone and generates user transaction graphs dynamically based on app instrumentation and online execution trace collection. These dynamic user transaction graphs (DUTGs) only capture the program execution flows for a specific user transaction and lack the complete picture of all possible execution flows in a user transaction. As a result, developers may miss important information, such as longer critical execution paths. In comparison, the authors of Panappticon [163] focus on Android and utilize lower level instrumentation in the operating system (OS) and Android framework to collect app execution traces, based on which DUTGs are constructed. However, such low-level instrumentation imposes a huge challenge to app developers, as Android framework is frequently updated. More critically, no prior work has systematically attempted to *capture all the possible execution paths* in a user transaction, including both *synchronous and asynchronous* execution paths, as well as the relevant *control flows* and utilize such knowledge to better serve the developers.

Overview of This Work. The goal of this work is to leverage *static analysis* to answer the questions above – *statically reasoning about the user transactions in mobile apps to discover all possible execution paths of each user transaction* and *utilizing the analysis results to carry out a variety of app profiling and testing tasks*, thus freeing developers from the tedious and error-prone app instrumentation in the presence of complex user interfaces and highly asynchronous executions.

In specific, we define **static user transaction graph** (SUTG)—a graph represents all possible temporal and causality relations among *operations of interest* during all possible user transactions triggered by a user manipulation. In its essence, SUTG summarizes the execution flows from the entry to all possible exit points of a user transaction, meanwhile captures all UI updates and start and synchronization of threads. To achieve our goal, we propose (i) a *static analysis* that reasons about app source code to generate SUTGs, and (ii) tools to automatically instrument apps using SUTGs for *user-perceived latency profiling* and *app testing in the presence of asynchronous state updates.*

In this work, we focus on the Android platform due to the popularity of platform (75% market share as of July 2020 [149]) and open-source ecosystem. Android apps consists of many different callbacks (i.e., event handlers) to respond to various events, generated due to user interaction (e.g., clicking, scrolling, and typing). These callbacks essentially serve as the starting points of user transactions. Our static analysis starts the construction of a SUTG from each user action callback. During the construction, the static analysis discovers interesting program points such as UI updates, asynchronous method calls, and inter-component communications (e.g., Service to Broadcast receiver communication), and connects them based on the causality relations as well as the control flows.

Once constructed, SUTGs can serve for different kinds of app testing and profiling tasks. In this work, we present two SUTG-based tools.

- *Automatic user perceived-latency profiler*, which leverages SUTGs to instrument mobile apps at the binary level and remove the instrumentation afterwards. It requires no app source-code level or OS/framework level modifications. Furthermore, the profiler can map the profiling results back to SUTGs to provide critical path reasoning and path coverage analysis to help developers find the performance bottlenecks.

- *Automatic state-management for asynchronous events*, whose subsequent operations affect a UI test, we refer as *idling resource manager*. Idling resource manager helps developers by automating idling resources API usages to carry out testing reliably when asynchronous executions are involved.

We evaluated static analyzer on popular open-source (from F-Droid [89]) and commercial (from Google Play [99]) apps, corpus of total 791 apps. Furthermore, we conducted a focused study on selected 44 apps (*GroupS*), and developed instrumentation tools for applications of SUTG called, LatencyInstrumentor and Idling-ResourceManager. In summary, this work makes the following contributions:

- We introduce *Static User Transaction Graph* based on possible flows of a user interaction, that captures the UI and control flow properties of a user transaction.
- We provide static analysis to automatically build Static User Transaction Graph for a given app, that developers can apply for the annotated or complete source code of their apps.
- We develop and provide two applications of Static User Transaction Graph, a user perceived latency profiler and an idling resources state manager.
- To evaluate Static User Transaction Graph and its applications, we applied our static analyzer on 791 Android applications from F-Droid [89] and Google Play [99], and show SUTG is applicable to real-world apps.

5.2 Static User Transaction Graph

In this section, we define static user transaction graph (SUTG) for mobile apps in detail, including information it captures and its structure, then we present the static analysis for constructing SUTGs.

5.2.1 Definitions

Definition 11 *A* **user transaction** ***u*** *starts with a user manipulation with the app* ***m*** *and ends when all* asynchronous *(threads) and* synchronous *operations triggered by* ***m*** *are completed.*

Roughly speaking, given a user transaction, a **dynamic user transaction graph** (DUTG) is a graph representing the temporal and causality relations among *operations of interest* during this specific user transaction. In comparison, a **static user transaction graph** (SUTG) is a graph representing all possible temporal and causality relations among *operations of interest* during all possible user transactions triggered by a user manipulation.

To facilitate a range of applications as discussed later, we define the SUTG such that it captures the following operations and relations of interest:

1. User interaction actions (e.g., a `Button` is clicked).

2. UI updates on user interfaces (e.g., text in `TextView` changes).

3. Causal relations between asynchronous calls and their callbacks (e.g., `AsyncTask`).

4. Synchronization points among threads (e.g., between an `AsyncTask` and UI thread).

5. Inter-component interactions (e.g., start a new `activity` for displaying results).

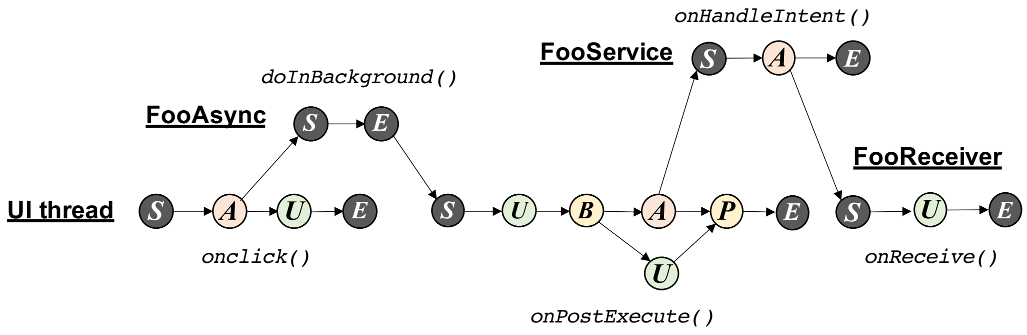


Figure 5.1: Static User Transaction Graph (SUTG) for the code snippets in Code 5.1 and Code 5.2.

6. Control flows relevant to the above operations and relations.

In the following, we discuss how SUTG captures the above operations in its structures.

Graph Structure. Roughly speaking, a SUTG can be viewed as a set of "simplified" control-flow graphs (CFGs) of callbacks, inter-connected by their *temporal causality* relations. Figure 5.1 shows an example SUTG for the code snippets in Codes 5.1 and 5.2.

A SUTG starts from a callback (i.e., event handler) that corresponds to the user action (e.g., `onClick()` at Line 2 in Code 5.1 for a user click). For each callback involved in the user transaction, the SUTG defines an ***S*** node and a ***E*** node, denoting the start and end of the callback, respectively. If the callback involves UI update operations (e.g., `bodyText.setText()` at Line 4 and 12 in Code 5.1 and Line 22 in Code 5.2), the SUTG defines a ***U*** node for each consecutive sequence of UI updates. When the callback spawns asynchronous threads (e.g., `AsyncTask().execute()` at Line 3 in Code 5.1) or performs inter-component communications (e.g., `sendBroadcast(intent)`) at Line 14 in Code 5.2, an ***A*** node is created for each call site,

Table 5.1: Node Types in SUTG.

Node Type	Description
S	start node of a callback
E	end node of a callback
B	a control-flow branching point
P	a control-flow joining point (a PHI-node)
U	UI updates
A	asynchronous callback invocation

meanwhile a directed edge is created from the ***A*** node to the ***S*** node of the asynchronous callback. For conciseness, in each callback, only the relevant control flows—control flows that connect the above defined nodes, are preserved in the SUTG. To separate nodes on different branches, SUTG defines ***B*** nodes and ***P*** nodes for branching and joining, respectively. For synchronous calls in the callback, SUTG "inlines" the above nodes (excluding ***S*** and ***E*** nodes) from the callees into the control flows of the caller, along with the relevant control flows. In the presence of recursive calls, a back edge is inserted into the SUTG to represent the repetitions of nodes within the recursive methods. Finally, all SUTG nodes encode the direct caller's information, so that they can be quickly located in the source code. Table 5.1 summarizes the node types of SUTG.

Based on the above definition, we can easily define a DUTG, which is a subgraph of SUTG such that only branch is taken at each ***B*** node in the SUTG.

5.2.2 Static Analysis for Graph Construction

Mobile apps use callbacks (i.e., event handlers) to respond to user actions. Our analysis starts the construction from these callbacks which are the *entry points* of user

```java
class FooActivity extends Activity{
  void onClick() {
    new FooAsync().execute();
    this.bodyText.setText("started!");
  }
  class FooAsync extends AyncTask {
    void doInBackground(){
      ... // local I/O to find data
    }
    void onPostExecute(){
      if(foundLocally)
        bodyText.setText("fetched locally!");
      else {
       filter = new IntentFilter("ACT_RESP");
       receiver = new FooReceiver();
       registerReceiver(receiver, filter);
       // start service
       service = new Intent(...,FooService.class);
       service.putExtra("request", url);
       startService(service);
      }
    }
  }
}
```

Code 5.1: Example Activity and AsyncTask .

```java
class FooService extends IntentService {
  public FooService() {
    super("FooService");
  }
  @Override
  void onHandleIntent(Intent intent) {
  String url = intent.getStringExtra("request");
  // fetch data from server based on url
  Data data = ... ;
  // send Broadcast
  Intent broadcastIntent = new Intent();
  broadcastIntent.setAction("ACT_RESP");
  broadcastIntent.putExtra("receive", data);
  sendBroadcast(broadcastIntent);
 }
}

class FooReceiver extends BroadcastReceiver {
  public void onReceive(Context c, Intent i) {
    Data data = i.getStringExtra("receive");
    // consume data
    bodyText.setText("fetched remotely!");
  }
}
```

Code 5.2: Example IntentService and BroadcastReceiver.

transactions. *The goal of our static analysis is to discover all possible synchronous and asynchronous exit points of a given callback and the UI operations along the paths reaching to the exit points, meanwhile encode the results in the form of a graph as defined above.* To achieve this, our analysis needs to address several complexities related to *UI updates*, *asynchronous calls* and *inter-component interaction*.

- ***Missing UI updates***. The state-of-the-art static analysis tools such as FlowDroid [76] fail to identify UI updates in Android apps, such as `bodyText.setText()` at Line 4 in Code 5.1, because the UI variables (like `bodyText`) are not necessarily defined in the Java source code explicitly (but in XML-based screen layout files).

- ***Asynchronous calls***. Some background threads spawned by asynchronous calls need to synchronize with the UI thread by invoking subsequent callbacks that run on the UI thread. This synchronization defines the causality relations between the asynchronous callback and its subsequent callback on the UI thread. However, the causality relations are not defined by the app's source code, instead, they are defined by the underlying Android framework, thus cannot be discovered by any app-level code analysis.

- ***Inter-component communication***. Nontrivial Android apps typically consist of multiple components of different types (such as `Activity`, `Service`, `BroadcastReceiver`, etc.), which may run concurrently with each other and communicate through passing instances of `Intent`. These inter-component communications define the causality relations among the senders and the receivers. For direct "point-to-point" communication, the causality can be easily discovered as the receiver is explicitly mentioned when sending the intent. However, for "broadcasting"-style communication (like `BroadcastReceiver`), receivers are absent in the intent sending call site (e.g., `sendBroadcast(broadcastIntent)` at Line 14 in Code 5.2), thus it is non-trivial to statically map a sender with its corresponding receiver(s).

In the following, we present the basic ideas of our static analysis, meanwhile address the above complexities. At high level, we design the analysis as a worklist-based iterative algorithm.

Initialization. To begin with, our analysis first collects the entry points of user transactions—the callbacks defined for user actions. For Android apps, callbacks for user actions can be registered either in Java source code or in the XML layout files. Our analysis scans both places to collect the list of callbacks registered for handling user actions, denoted as $Callbacks_{user}$. For each user action callback, our analysis creates a pair of $\boldsymbol{S}$ and $\boldsymbol{E}$ nodes, representing their initial SUTGs.

Iteration. During the iterations, our analysis maintains a worklist of callbacks $Callbacks_{worklist}$, which is initialized with the user action callbacks $Callbacks_{user}$. For each callback in the worklist, our analysis first constructs its inter-procedural control-flow graph (ICFG), then traverses its ICFG, meanwhile performs the following tasks:

- ***Identify UI update call sites.*** As mentioned earlier, UI update calls are missing in the ICFGs generated by the state-of-the-art analyzers. To solve this problem, our analysis first collects the references to UI variables—instances of various types of `View` class. This can be achieved by looking for the uses of `findViewById()`, which is an Android API that developers use to obtain view references. After that, our analysis generates explicit allocations for these collected UI variables in the Java source code.

In this way, ICFG generator (like FlowDroid) is able to generate ICFGs with UI update call sites. To identify the UI update call sites in the ICFGs, our analysis just needs to check the variable type (subclasses of `View` class) and the APIs (update APIs are pre-collected based on API documentations of different `View` classes). For developer-defined UI classes, their update APIs can also be appended to the list of update APIs. For conciseness, the analysis creates a single ***U*** node for each consecutive sequence UI update calls. The information regarding the update calls is stored as attributes in the ***U*** node.

- ***Identify asynchronous call sites and their callbacks***. Our analysis recognizes call sites for asynchronous executions, like `FooAsync().execute()` and their subsequent callbacks, like `doInBackground()` and `onPostExecute()`, based on offline modeling of the asynchronous constructs collected from the Android and Java API documentations. For each asynchronous call site, our analysis creates an ***A*** node. Then, for each asynchronous callback, our analysis first creates a pair of ***S*** and ***E*** nodes, then adds it into the callback worklist $Callbacks_{worklist}$. Finally, our analysis connects the ***A*** node to the ***S*** node of the first callback, and chains the callbacks by adding an edge from a ***E*** nodes to the ***S*** node of the followup callback.

- ***Identify inter-component communications***. Different instances of Android components in an app (like `Activity`, `Service`, and `BroadcastReceiver`) communicate by passing `intent` instances. The key to identify inter-component communications is to find the intent filters which pairs the sender component with the receiver component. The filters can be easily located based on their definitions (like the `filter` at Line 14 in Code 5.1). From the filters, our analysis extracts the filter rules (e.g., `ACT_RESP`), based on which it locates the components and methods that use the filters for sending intents (e.g., `boradcastIntent.setAction(ACT_RESP)` at Line 12 in Code 5.2). For the intent sending call site, our analysis creates an ***A*** node. Then, based on the filter registration information (e.g., `registerReceiver(receiver, filter)` at Line 16 in Code 5.1), our analysis finds the receiver component (e.g., `FooReceiver` at Line 18 in Code 5.2). Depending on the component types of the receiver, our analysis creates a single wrapper callback that encloses all the lifecycle callbacks of the receiver component that are invoked during its creation and initialization (e.g., `onReceive()` at Line 19 in Code 5.2). Then, our analysis creates a pair of ***S*** and ***E*** nodes for the wrapper callback, then add it to the callback worklist. Finally, our analysis connects the ***A*** node representing the intent sending call site to the ***S*** node of the wrapper callback of the intent receiver.

- ***Identify relevant control flows***. During the ICFG traversal, once our analysis finds control flows, such as branching and joining points, that are relevant to the identified UI update call sites, asynchronous call sites, and intent sending call sites, it creates ***B*** nodes and ***P*** nodes correspondingly in the SUTG. In specific, when the analysis discovers a control-flow branching point, if there exists at least one of the above call sites of interest on at least one of its branches, it creates a ***B*** node representing the branching point in the SUTG. Correspondingly, when the analysis reaches a control-flow joining point, if there exists a path from the branching point to the joining point along which one of the above call sites of interest occurs, then the analysis creates a ***P*** node representing the joining point.

To generate inter-procedural control-flow graph (ICFG), we leverages FlowDroid [76]. To keep our graph concise, we only encode the direct caller's information in a SUTG node.

5.3 Applications of Static User Transaction Graph

Once SUTGs are constructed, they can be utilized for different use cases such as profiling and testing of the apps. In the following, we present two applications of SUTGs.

5.3.1 User-Perceived Latency Profiling

It is important for developers to improve the performance of their apps, especially user-perceived latency during the interactions. For simple scenarios, such as changing some text on the screen after a user clicks, developers can easily instrument the event handlers of the user actions to measure the latency. However, in presence of asynchronous executions, measuring the time spent for a user transaction could be complicated, since the developers need to consider the concurrent executions of different callbacks and their causality relations. To free developers from the above tedious and complex instrumentation, we propose to use SUTGs to automatically instrument the apps, which then collect the latencies of different user transactions during the app testing.

Specifically, we provide a lightweight runtime client (in code 5.3) that generates log messages with timestamps and unique SUTG node identifiers. The generated logs provide information to measure the latency among the nodes of a given SUTG and derive the total latency of a user transaction. Other than timestamp information, it is also important to

```java
public class LatencyProfiler{
  enum State {IDLE, START, RUNNING, STOP};
  private static LatencyProfiler instance;
  private final String TAG = "LatencyProfiler";
  private HashMap<String, State> eventStateMap; // event name and its status
  private HashMap<String, Set<String>> backgroundTasks; // set of background tasks
  public void registerEvents(String annotationInfo, String[] events) {
    eventStateMap.put(annotationInfo, State.IDLE);
    Set<String> tasks = new HashSet<>();
    for (String event : events) {
      if (tasks.add(event)) this.eventStateMap.put(event, State.IDLE);
    }
    this.backgroundTasks.put(annotationInfo,tasks);
  }
  void reportProgress(String annotationInfo, String event, State eventState) {
    Log.i(TAG, annotationInfo + ":" + event + ":" +getTimeInMillis());
    this.eventStateMap.put(eventName, eventState);
    boolean mainThreadCompleted = isMainCompleted(annotationInfo);
    boolean anyAsyncTaskInProgress = isAnyAsyncTaskInProgress(annotationInfo);
    if(mainThreadCompleted && !anyAsyncTaskInProgress){
      //User Transaction Completed, remove event
      Log.v(TAG, annotationInfo+":transactionFinished:"+getTimeInMillis());
      this.eventStateMap.remove(annotationInfo);
      this.backgroundTasks.remove(annotationInfo);
    }
  }
}
```

Code 5.3: Runtime Module for Latency Profiler.

```
class FooActivity extends Activity{
  ...
  @LatencyProfiler(info = "btnClick")
  void onClick() {
    LatencyProfiler.getInstance().registerEvents("btnClick", new String[0]);
    LatencyProfiler.getInstance().reportProgress("btnClick", "MAIN", State.START);
    this.bodyText.setText("Text Updated!");
    LatencyProfiler.getInstance().reportProgress("btnClick", "MAIN", State.STOP);
  }
}
```

Code 5.4: Example of simple User Transaction with Latency Profiling Instrumentation.

keep track of ongoing asynchronous operations at runtime, since an asynchronous operation can continue running while its trigger has been finished already, or another user action can cause to start it while an existing instance is in-progress. We keep track of ongoing user transactions along with relevant in-progress asynchronous operations, which can be achieved by maintaining a couple of HashMap data structures, as shown in Code 5.3. First HashMap (`eventStateMap`), keeps track of ongoing user transactions, and another HashMap (`backgroundTasks`) maintains the state of associated asynchronous executions for the user transactions in `eventStateMap`.

We show the usage of our client in code 5.4 and code 5.5 for simple examples, first, we need to register potential asynchronous operations for a user action and then keep reporting the progress of user transactions. When all the ongoing synchronous and asynchronous operations associated with a user transaction has finished, then, `LatencyProfiler` class will report that the user transaction has been completed.

5.3.2 Idling Resource Manager

As mentioned in Section 2.4, the default UI testing tool for Android apps, Espresso [61], could provide reliable synchronization when an asynchronous execution could update the views (referred to as *Idling Resource*). However, developers need to include the implementation of the `CountDownLatch` API, which is used by Espresso for synchronization. To utilize this, first, developers need to figure out synchronization logic and then integrate `CountDownLatch` API calls carefully at correct places, which could be challenging, especially when multiple asynchronous executions are involved within a single user transaction. Secondly, as developers prepare the release version, they need to remove the code added in the first step (i.e., synchronization logic), and they will need to perform this addition and removal every time they test and release a new version of the app. Code 5.6 shows an example of `CountDownLatch`, app, first increments the counter using `idlingResource.increment()` API, and on completion perform decrement by calling `idlingResource.decrement()`. Finally, an idling resource is considered idle, when "count" is at zero. For example, for 10 spawn threads, first `idlingResource.increment()` will be called at the start of each thread, and `idlingResource.decrement()` will decrement the counter when the thread finishes, eventually, when the counter reaches zero (i.e., idle state), it means all threads are finished and Espresso test can continue the test case to perform checks on view states.

```java
class FooActivity extends Activity{
  @LatencyProfiler(info = "btnClick")
  void onClick() {
    String[] events = {"btnClick:TestAsync"};
    LatencyProfiler.getInstance().registerEvents("btnClick", events);
    LatencyProfiler.getInstance().reportProgress("btnClick", "MAIN", State.START);
    new TestAsync().execute();
    this.bodyText.setText("Text Updated!");
    LatencyProfiler.getInstance().reportProgress("btnClick", "MAIN" ,State.STOP);
  }
  class TestAsync extends AyncTask {
    void onPreExecute(){
        LatencyProfiler.getInstance().reportProgress("btnClick",
             "btnClick:TestAsync" ,State.START);
        bodyText2.setText("Started Text");
    }
    void doInBackground(){
      ... // no UI access allowed
    }
    void onPostExecute(){
        bodyText2.setText("Completed Text");
        LatencyProfiler.getInstance().reportProgress("btnClick",
             "btnClick:TestAsync" ,State.STOP);
    }
  }
}
```

Code 5.5: Example of simple User Transaction with AsyncTask with Latency Profiling Instrumentation.

```
class FooActivity extends Activity{
  ...
  void onClick() {
    ...
    processMessage(getIdlingResource());
  }
  @AsyncAssertion(info = "processMessage")
  static void processMessage(CountingIdlingResource idlingResource) {
    idlingResource.increment();
    // Asynchronous execution
    Handler handler = new Handler();
    handler.postDelayed(new Runnable() {
      @Override
      public void run() {
        ...
        idlingResource.decrement();
      }
    }, DELAY_MILLIS);
  }
  @VisibleForTesting
  public CountingIdlingResource getIdlingResource() {
    if (mIdlingResource == null) {
      mIdlingResource = new CountingIdlingResource("processMessage");
    }
    return mIdlingResource;
  }
}
```

Code 5.6: Example of Automatic assertions for Asynchronous Events.

```
@RunWith(AndroidJUnit4.class)
public class FooActivityTest {
  ...
  @Before
  public void registerIdlingResource() {
    ActivityScenario activityScenario =
        ActivityScenario.launch(FooActivity.class);
    activityScenario.onActivity(new
        ActivityScenario.ActivityAction<FooActivity>() {
      @Override
      public void perform(FooActivity activity) {
        mIdlingResource = activity.getIdlingResource();
        IdlingRegistry.getInstance().register(mIdlingResource);
        }
    });
  }
  @After
  public void unregisterIdlingResource() {
    if (mIdlingResource != null) {
      IdlingRegistry.getInstance().unregister(mIdlingResource);
    }
  }
}
```

Code 5.7: JUnit Example of Automatic testing for Asynchronous Events.

Code 5.7 shows a test class example for idling resource, developers first need to get `IdlingResource` instance from the relevant activity and register with the Espresso testing environment using `IdlingRegistry.register()` API call (appearing in `registerIdlingResource()` method). The registration with the testing framework is usually performed before starting the test, and upon finishing the test case, the registered idling resource is unregistered using `IdlingRegistry.unregister()` API call.

In general, the test cases, registering, and un-registering the idling resource remain app-specific and also developers do not need to remove them before releasing the app. Thus, this work does not go into that and focuses on maintaining the synchronization logic of idling resource, which developers need to add for testing and remove at the time of release. By utilizing SUTG, as it encodes information about the trigger and synchronization of asynchronous executions, we can instrument apps to add testing code with `CountDownLatch` API calls, consequently, it will free developers from adding and removing synchronization logic manually, when testing asynchronous executions.

5.4 Implementation

As some of the technical details about SUTG analysis and applications of SUTG have been discussed already, in this section, we focus more on static analysis and instrumentation tools that realize the SUTG analysis and its applications. SUTG-Analyzer implements the SUTG analysis, then based on the analysis results, both LatencyInstrumentor and Idling-ResourceManager instrument the APK files for applications of

SUTG. Furthermore, we provide a tool to annotate the user actions that enables developers to apply static analysis and instrumentation for selected user actions only.

SUTG-Analyzer. The static analyzer, namely SUTG-Analyzer, is built on top of Soot [144] program analysis framework. SUTG-Analyzer works in two modes, (i) *Process-All* – to process all the user actions defined in the app, and (ii) *Process-Annotated* – which processes the only annotated user actions (i.e., event handler methods). The other libraries that SUTG-Analyzer rely on include, Spark [118], FlowDroid [76], and Backstage [77]. The intermediate representation (Jimple) is provided by Soot [144] which facilitate us to analyze Java programs. SUTG-Analyzer takes an app executable (APK) as input and uses FlowDroid [76] to generate inter-procedural control-flow graphs (ICFGs), FlowDroid also collects all the user actions when SUTG-Analyzer is used in *Process-All* mode. Backstage [77] resolves the target Android components when those components are invoked via Android's `Intent` APIs. SUTG-Analyzer also takes in a list of `View` widgets, that developers can utilize to add their custom UI widgets and our analysis would include them in the analysis. Finally, SUTG-Analyzer outputs the aggregated results along with graph description language (.dot extension) [29] file for each SUTG, which provides visualization of SUTGs.

Method Annotation. As discussed earlier, developers can use SUTG-Analyzer in *Process-Annotated* mode to utilize this option, developers can use `@LatencyProfiler` and `@IdlingResourceManager` annotations for user-perceived latency profiling and idling resources manager, respectively. Later on, when analyzing, SUTG-Analyzer can identify annotations and perform analysis for only annotated user actions (i.e., Java methods).

APK Instrumentation Tools. We also provide APK instrumentation tools for *Latency Profiling* and *Idling resource management*, named LATENCYINSTRUMENTOR and IDLING-RESOURCEMANAGER, respectively. These tools take in an APK file and result from SUTG-ANALYZER as input and insert the code based on SUTG nodes as discussed earlier. We use Soot [144] for code-generation, reverse-engineering, and recompilation. After that, we rely on Android SDK tools such as `Zipalign` [44] and `apksigner` [16] to align and sign the final APK. Finally, our instrumentation tools produce and sign the APK file that developers can install, and run to profile and test.

Post-processing. The instrumented app using LATENCYINSTRUMENTOR generates log messages using Android's `Log.v()` API at runtime, these log messages are collected and written on the developer's machine using our post-processing tool. We use Android Debug Bridge (adb) [143] and `logcat` tools from Android SDK, specifically, we filter out our relevant messages using `adb logcat -s "LatencyProfiler"`, where "LatencyProfiler" is our tag that we use in latency profiler as discussed earlier. At the end of the testing session, all the collected logs are processed and analysis results are generated, along with annotated-SUTGs. For instrumented apps using IDLING-RESOURCEMANAGER, espresso tests can be directly performed after installing the APK using post-processing tool, which utilizes activity manager and instrumentation monitoring (`am instrument`) from Android SDK .

Limitations. Although SUTG-ANALYZER aims for an accurate solution, the static analysis inherits limitations from other static analysis tools. For example, the stable version of FlowDroid [76] used in SUTG-ANALYZER does not support lambda-style event declarations in Java 8 and native method modeling. It also inherits limitations on reflective calls, which are resolved only if their arguments are string constants. The SUTG-Analyzer analysis can only process Java code at the moment, as Soot does not fully support the `invokedynamic` bytecode [93] which affects apps written in Kotlin (or Java code using lambdas and method refs). Similar limitations apply to the plug-in, which is written for Java only, and cannot process `NativeActivity` [57] or apps written in Kotlin.

5.5 Evaluations

This section evaluates SUTG-Analyzer on real-world Android apps to demonstrate its applicability, effectiveness in generating the SUTGs. Furthermore, we utilize SUTGs for applying LatencyInstrumentor and Idling-ResourceManager and measure their performance and correctness of instrumentation of APK files.

5.5.1 Methodology

To evaluate SUTG-Analyzer, we crawled 800 apps from F-Droid [89] and 400 top apps from Google Play [99], and retained 535 apps from F-Droid (denoted as *GroupF*) and 256 Google Play apps (denoted *GroupP*). The removed apps either do not contain activities (potentially do not offer user actions), or FlowDroid failed to build an Interprocedural Control-Flow Graph (ICFG). We selected an additional set of 44 apps denoted as *GroupS*, these apps are from F-Droid and Google Play with at least 30 stars on GitHub [94] in case of open-source apps, and more than one million downloads on Google Play. As shown in table 5.3, they include some highly influential open-source projects such as NewPipe and

Free RDP, and some widely used commercial apps including BBC News, Wall Street Journal and Alaska Airlines.

Collectively, *GroupF* consists of 3,424 Android components, *GroupP* has 5,627 Android components, and 1,164 Android components are in *GroupS*. Note that, we refer to activities, services, broadcast receivers, and contents providers as Android components. We use *GroupF* and *GroupP* applicability of static analysis in general and *GroupS* for a focused study including statistics on SUTGs, and time cost for generating SUTGs. For each app, we applied SUTG-Analyzer on each component registered in the `AndroidManifest.xml`.

To examine the actual application behaviors when LatencyInstrumentor and Idling-ResourceManager applied, we use a Nexus 5X smartphone running Android version 8.1. The programming environment is Android Studio 3.4. Experiments on *GroupF* and *GroupP* were conducted on a PC with a 3.5 GHz Intel Xeon processor and 16 GB RAM, while results for *GroupS* were collected on a MacBook Pro with 1.4 GHz Quad-Core Intel Core i5 processor and 8 GB RAM.

5.5.2 Static Analysis

We evaluated the applicability of SUTG-Analyzer on both *GroupF* and *GroupP*, and the results of static analysis on *GroupS* in detail, including performance, statistics of SUTGs, and its correctness. Overall, our results show that SUTG-Analyzer applies to various real-world apps and generates SUTGs covering all the possible flows of user

Table 5.2: Results of running SUTG for *GroupS* and *GroupP*.

	GroupF	*GroupP*
Total Apps	535	256
Total Android Components	3,424	5,627
Total Event Handlers	8,007	11,606
Total Nodes in SUTGs	42,634	59,980
Total Paths in SUTGs	13,791	19,776
# of Events with multiple Paths	2,362	3,078
# of async Paths	1,077	2,057
# of async Nodes	2,814	5,264
# of Branches (Decisions)	8,801	11,899
# of UI Nodes	6,049	8,444
# of Apps with async path	207 (38.6%)	128 (50.0%)

transactions. In the following, we first present the results from the applicability of SUTG-Analyzer, then discuss the detailed results of static analysis on *GroupS*.

Applicability. The summary of static analysis results is reported in table 5.2 for *GroupF* and *GroupP*. At the application level, among *GroupF* apps, 38% (207 out of 565) apps, and 50% of apps (208 out of 256) from *GroupP* contain at least one asynchronous path, which shows that a naive profiler would fail to measure user-perceived latency due to potential asynchronous executions, and also these apps would require manual effort to implement idling resource management during testing. As to the complexity of generated SUTGs, there are 42,634 and 59,980 nodes in generated SUTGs for *GroupF* and *GroupP*, respectively. While we did not observe any failures during the above analysis, there are a couple of situations where the SUTG-Analyzer may fail, including `NativeActivity` written in C/C++ and activities implemented in Kotlin (Soot does not fully support `invokeDynamics` [93]).

Time Cost. We measure the time cost to generate SUTGs for GroupS and report the results in Table 5.4 under column "Time". For most apps (40/44), the analysis finishes within 1 minute and often within 10 seconds (with an arithmetic mean of 29 seconds).

Table 5.3: Detailed static analysis results from applying SUTG-Analyzer on *GroupS*.

Stars: #stars on GitHub, Installs: #downloads on Google Play,

AC: #Android components, EH: #event handlers analyzed.

#	Package	Stars	Installs	AC	EH
1	com.bytestemplar.tonedef	49	10K+	9	12
2	com.eleybourn.bookcatalogue	345	100K+	22	143
3	com.freerdp.afreerdp	5,603	-	10	46
4	com.futurice.android.reservator	177	-	6	13
5	com.ihunda.android.binauralbeat	152	1M+	1	23
6	com.jadn.cc	57	100K+	16	51
7	com.mschlauch.comfortreader	30	1K+	5	20
8	com.nightonke.cocoin	2,734	-	18	154
9	com.sagar.screenshift2	50	1M+	10	41
10	com.ubergeek42.WeechatAndroid	460	-	9	10
11	com.zzzmode.appopsx	439	-	11	20
12	de.k3b.android.toGoZip	40	-	3	22
13	douzifly.list	58	-	6	49
14	io.github.hidroh.materialistic	2,152	100K+	38	57
15	io.github.pd4d10.gittouch	897	1K+	3	18
16	jackpal.androidterm	2,624	10M+	9	40
17	org.billthefarmer.diary	174	10K+	4	25
18	org.glucosio.android	332	-	17	76
19	org.mozilla.klar	1,835	500K+	12	12
20	org.schabi.newpipe	15,063	-	17	55
21	org.sipdroid.sipua	456	-	21	48
22	org.xbmc.android.remote	366	-	25	145
23	bbc.mobile.news.ww	-	10M+	33	32
24	cc.pacer.androidapp	-	10M+	122	229
25	cl.datacomputer.alejandrob.newgpsjoystick	-	1M+	20	42
26	com.alaskaairlines.android	-	1M+	75	112
27	com.aws.android.tsunami	-	1M+	24	14
28	com.bskyb.fbscore	-	10M+	25	57
29	com.campmobile.snow	-	100M+	60	116
30	com.exutech.chacha	-	10M+	50	114
31	com.fundevs.app.mediaconverter	-	100M+	35	66
32	com.geekslab.qrbarcodescanner.pro	-	10M+	12	46
33	com.homesnap	-	1M+	105	187
34	com.jimmyjohns	-	1M+	10	30
35	com.magix.android.mmjam	-	10M+	29	112
36	com.musicplayer.music	-	50M+	28	94
37	com.secretdiaryappfree	-	5M+	23	72
38	com.showmax.app	-	10M+	23	88
39	com.synergetechsolutions.nearbylive	-	5M+	39	88
40	com.uc.browser.en	-	100M+	29	114
41	jp.co.rcsc.yurekuru.android	-	1M+	27	102
42	net.metapps.sleepsounds	-	5M+	10	31
43	uk.co.economist	-	1M+	67	135
44	wsj.reader_sp	-	1M+	46	74

Table 5.4: Detailed static analysis results from applying SUTG-Analyzer on *GroupS*.
Nodes: #nodes in SUTGs, Paths: #program flow paths in SUTGs,
Async: #asynchronous paths in SUTGs, UI: #UI nodes in SUTGs, Time: analysis time (ms).

#	Package	Nodes	Paths	Async	UI	Time
1	com.bytestemplar.tonedef	65	17	7	6	4,263
2	com.eleybourn.bookcatalogue	642	211	21	109	16,663
3	com.freerdp.afreerdp	196	70	8	14	10,872
4	com.futurice.android.reservator	90	26	8	18	2,903
5	com.ihunda.android.binauralbeat	59	29	0	3	2,054
6	com.jadn.cc	238	72	6	48	3,217
7	com.mschlauch.comfortreader	125	31	6	25	8,931
8	com.nightonke.cocoin	1,053	247	32	248	53,342
9	com.sagar.screenshift2	302	129	15	41	7,762
10	com.ubergeek42.WeechatAndroid	40	15	1	6	12,725
11	com.zzzmode.appopsx	105	34	2	2	10,270
12	de.k3b.android.toGoZip	124	34	6	15	4,098
13	douzifly.list	386	112	18	67	11,376
14	io.github.hidroh.materialistic	218	79	1	5	13,711
15	io.github.pd4d10.gittouch	65	20	0	7	3,270
16	jackpal.androidterm	223	68	1	32	4,602
17	org.billthefarmer.diary	139	36	0	24	3,643
18	org.glucosio.android	404	120	19	55	17,926
19	org.mozilla.klar	56	15	0	3	12,066
20	org.schabi.newpipe	224	61	1	24	21,399
21	org.sipdroid.sipua	541	494	22	36	86,908
22	org.xbmc.android.remote	1,340	451	154	176	15,024
23	bbc.mobile.news.ww	180	44	4	10	42,373
24	cc.pacer.androidapp	1705	595	190	107	131,401
25	cl.datacomputer.alejandrob.newgpsjoystick	268	81	4	29	40,110
26	com.alaskaairlines.android	559	194	24	19	35,306
27	com.aws.android.tsunami	71	28	1	12	17,797
28	com.bskyb.fbscore	270	91	5	25	31,187
29	com.campmobile.snow	537	209	11	94	41,611
30	com.exutech.chacha	609	223	31	58	125,499
31	com.fundevs.app.mediaconverter	311	121	5	30	26,341
32	com.geekslab.qrbarcodescanner.pro	333	131	24	31	14,267
33	com.homesnap	878	287	24	108	62,753
34	com.jimmyjohns	200	39	8	13	17,276
35	com.magix.android.mmjam	817	333	54	71	52,145
36	com.musicplayer.music	722	229	36	125	33,999
37	com.secretdiaryappfree	438	111	20	79	23,593
38	com.showmax.app	469	133	6	74	54,368
39	com.synergetechsolutions.nearbylive	520	170	28	71	41,431
40	com.uc.browser.en	699	230	12	85	23,253
41	jp.co.rcsc.yurekuru.android	582	210	32	124	47,644
42	net.metapps.sleepsounds	192	62	11	29	6,625
43	uk.co.economist	636	202	15	54	38,519
44	wsj.reader_sp	333	110	6	27	42,413

App#24 took the most time (i.e., 131 seconds), analysis result reveals that this app contains a higher number of asynchronous and Android components compared to other apps. The presence of Android components along with asynchronous execution paths increase the analysis workload.

SUTG Nodes Statistics. Table 5.4 also reports different statistics about analysis results. Column "Nodes" reports the number of nodes in SUTGs. We found that 30 out of 44 apps have more than 200 nodes in their SUTGs. The column "Paths" reports control-flow paths in SUTGs, results reveal that 32 apps from *GroupS* contain more than 50 paths. Additionally, we found that 33.9% of SUTGs have multiple paths. We also report the number of asynchronous paths for SUTGs in column "ASYNC", we found 41 out 44 apps have at least one asynchronous path, and 28.9% of SUTGs contains at least one asynchronous execution on their paths. Among them, app#8 and app# 22 have the highest number of asynchronous paths, these apps make several network requests using `AsyncTask` construct, and display that inside the app UI, which results in a higher number of UI nodes for these apps in column UI. Overall, the UI column shows that all the apps have UI nodes, which is expected in mobile apps due to their highly user-interactive nature.

5.5.3 Applying SUTG for Latency Profiling and Testing

In the following, we first evaluate the applicability of code generation for latency profiling and idling resource manager by applying them on APK files. We report the time and space costs of code generation, as well as the runtime costs of `LatencyProfiler`. For the applicability evaluation, we use apps in *GroupS*; For the cost measurements, we randomly select 20 apps from *GroupS* as they involve significant manual efforts.

Applicability. First, we generated the SUTGs by applying SUTG-Analyzer for each app in *GroupS*, then we loaded its APK in LatencyInstrumentor and inserted `LatencyProfiler` class and API calls for `LatencyProfiler`. We confirmed that the LatencyInstrumentor was applied successfully, and new APK files for all the apps have been generated. To instrument apps for idling resources management, it is necessary to have test cases for apps available, unfortunately, the APKs in *GroupS* do not include test cases. As an alternative, we tested this instrumentation on a Micro-benchmark consisting of 5 apps and 3 sample apps from Google (referred to as Google-benchmark). The apps in Micro-benchmark are written by us and Google samples are available at GitHub [94], all apps include espresso test cases and asynchronous executions. We tested the Idling-ResourceManager for all the apps in both Micro-benchmark and Google-benchmark and generated new APK files. Again, we did not observe any issues when using the tool. These results demonstrate the applicability of our developed tools.

Time and Space Cost for Instrumentation. Table 5.6 presents the time and space cost of applying LatencyInstrumentor on selected apps from *GroupS*. LatencyInstrumentor took an average of 13.4 seconds, with the a maximum of 30 seconds for the “BBC News” app, to perform instrumentation based on the SUTG-Analyzer results. Columns “Original APK” and “Instrumented APK” compare the size of apps before and after instrumentation in kilobytes, and instrumented binaries result in a 2.7% increase in size due to the code injection to existing APK files.

Table 5.5: Time (ms) and space (Kilobytes) costs of instrumentation using LatencyInstrumentor.

Package	Original APK	Instrumented APK	Time
com.bytestemplar.tonedef	974	1,043	4,288
com.eleybourn.bookcatalogue	640	659	4,243
com.freerdp.afreerdp	20,675	20,709	12,291
com.sagar.screenshift2	2,514	2,563	7,610
com.ubergeek42.WeechatAndroid	1,994	2,036	7,472
com.zzzmode.appopsx	1,555	1,682	8,558
douzifly.list	1,766	1,983	7,849
io.github.hidroh.materialistic	3,259	3,420	16,584
io.github.pd4d10.gittouch	26,884	26,926	7,446
jackpal.androidterm	555	575	2,687
bbc.mobile.news.ww	12,911	13,292	30,979
com.geekslab.qrbarcodescanner.pro	2,188	2,371	17,162
com.overstock	7,415	8,269	29,288
uk.co.economist	8,362	8,552	18,429
wsj.reader_sp	7,637	8,029	27,210
Arithmetic Mean	6,622	6,807	13,473

For Idling-ResourceManager, we applied instrumentation on all the apps in Micro-benchmark and Google-benchmark, and we found that on average instrumentation takes 4.9 seconds. This instrumentation is much lighter than the instrumentation performed using LatencyInstrumentor, as Idling-ResourceManager requires instrumentation for asynchronous operations only.

Case Study on User-perceived latency Profiling. We randomly selected 20 apps to measure user-perceived latency using our tools. The apps were instrumented using LatencyInstrumentor based on generated SUTGs and deployed on Nexus 5X, then we used each app from the user's perspective with informal use cases, some of the use cases are listed in table 5.6. We used each app for 10 minutes, while attached to the development machine, and collected the profiling results (using ADB Logcat). Column "Use Case" shows

Table 5.6: Runtime profiling after applying LatencyInstrumentor.

Package	Use case	SUTG	DUTG	Memory	T_t	T_o
io.github...materialistic	Add Bookmark	7	4	1,134	28	0.11
	Remove Bookmark	7	4	1,134	37	0.11
	Up vote a comment	6	4	1,491	12	0.11
	Open news details	7	4	1,134	42,188	0.11
jackpal.androidterm	Take wifi Lock	7	5	695	6	0.13
	Drop Wifi Lock	7	5	695	7	0.13
	Take Wake Lock	7	5	687	9	0.13
	Drop Wake Lock	7	5	687	8	0.13
	Switch Session	8	6	582	53,874	0.16
bbc.mobile.news.ww	Open news details	22	13	1,003	25,356	0.35
	Search a news topic	14	8	1,005	524	0.22
com.overstock	Open Wish List	10	8	515	28,309	0.22
com.geekslab...pro	Generate QR code	20	16	824	118	0.43
wsj.reader_sp	News feed	24	9	827	16,185	0.24

the user action taken in the app, which we execute during testing. The number of nodes in SUTG and number of visited nodes is shown in Columns "SUTG" and "DUTG", respectively. We also measured the memory overhead caused due to our profiler at runtime, which is shown in column "Memory" in bytes, we observe that overhead remains usually within 1 Kilobyte and maximum usage observed is up to 1,491 bytes. Note that, memory overhead does not include log buffer, as we do not keep the log in the memory, logs are transferred to the development machine instantly via ADB and written to a file. Columns T_t and T_o report latency time for user transactions and runtime overhead due to our instrumentation in milliseconds results clearly show that our instrumentation remains low-overhead. Based on our measurements for one million operations for `Log.v()` API – Android SDK provided logger, which we also utilize for logging, costs 20 μs (microseconds). The cost for single-use of our API remains 27 μs, which is 7 μs more than `Log.v()` API usage, is due to operations performed on HashMap data structures to keep track of ongoing user transactions and their relevant asynchronous operations.

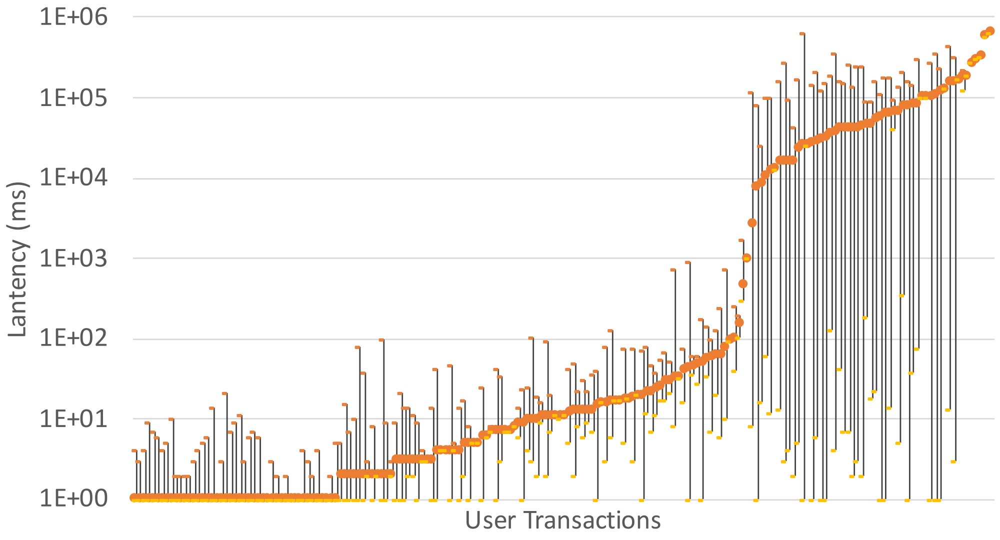


Figure 5.2: Length (in milliseconds) of User Transactions of selected apps.

After using each app, we report the average length of user transactions in Figure 5.2. As expected, we observed the majority of user transactions finished within a shorter period, whereas some of the user transactions could vary from a few seconds to minutes. After more analysis, we found when a user action takes an asynchronous path, which results in a longer latency. One key reason for such variation in latency is local caching performed by the apps, which reduces latency significantly after the first execution of a user transaction.

Improving coverage in Firefox Focus app. During post-processing of profiled user actions, we also provide the list of missed event handlers that did not execute during the profiling session. Developers can utilize this information to increase the coverage to test all the available user actions in the app. For example, When we profiled instrumented version of *Firefox Focus* [62] (an open-source app), after profiling we found that we missed the following three users actions during the first session.

1. `onLongClick()` from class `LinkHandler`, which responds to long click user action on browser address bar.

2. This app allows to add custom URL completion, when a user adds a such entry, the event handler `onSharedPreferenceChanged()` in `UrlMatcher` class gets triggered.

3. Finally, `onDownloadStart()` in `SystemWebView` class invokes when the app starts downloading a file.

At the end of the session, our post-processing tool pointed out these events (method signatures with class names). Based on that we were able to see user actions required for these events from the app source code and we perform these actions in the next session.

Another way to improve profiling coverage is to report paths that were not visited during the testing, for example, starting a download can take several paths within `onDownloadStart()` method. When we perform a download and execution completes, the post-processing tool reports that a path is never executed. This information is available on annotated SUTG visualization in Dot file as well, we can generation annotated SUTG based on static analysis SUTG and collected information at runtime. We further looked into the application code and found that Firefox focus allows downloading files over HTTP and HTTPS only, when downloading a file other than these protocols the missed path will be taken. Next time, we tried to download a file over FTP (which is not supported), downloading failed, and as a result, the missed path was executed.

Correctness of idling resource management. As mentioned earlier, for idling resource management, execution of test and synchronization is managed by Espresso testing environment and our state management is limited to correctly managing the logic and executing count latch API for idling resources, which is required to carry-out testing correctly for asynchronous components of the apps. We evaluated idling resource manager on Micro-benchmark and Google-benchmark, after applying it on both benchmarks, we executed tests, and manually verified the correctness of instrumentation.

5.6 Conclusion

This paper presents static user transaction graph (SUTG), which captures critical temporal and causality relations among concurrently executed callbacks during the user transactions. SUTG can be automatically constructed by static analysis and be leveraged as the basis for various mobile instrumentation and testing scenarios. In this work, we applied SUTG in two applications: automatically profiling user-perceived latency and managing idle resources for app state testing. The SUTG-based latency profiler requires no modifications to the underlying mobile OS, and can help developers find critical execution paths in user transactions. Both applications can free developers from tedious and error-prone manual instrumentation tasks. Finally, our evaluation using a wide range of real-world mobile apps from Google Play and F-Droid show that the SUTG-based tools can be successfully applied to real-world apps, provide accurate measurements, and requires little or no interventions from developers.

Chapter 6

Conclusions

This dissertation presents runtime system, static analysis, and developer tools to help mobile system designers and mobile app developers better manage the mobile state in volatile runtime environments. All the presented techniques have been examined and evaluated on real-world mobile apps and have demonstrated their effectiveness. Specifically, this dissertation has the following key contributions.

First, it presents 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.

Second, this dissertation targets a major challenge in developing reliable mobile apps – a volatile runtime environment that repeatedly destroys app state through activity and app-level restarts. The solution is an automatic approach that combines callback modeling, static analysis, and runtime data saving/restoring techniques. The callback modeling categorizes different callbacks based on their invocation orders relatively to the user interaction. The static analysis takes both app code (Java class) and GUI interface (layout file) as inputs and identifies critical external and internal app state via novel, necessary instance state, abstraction. The runtime module saves and restores the identified app state, based on the actual aliases. Finally, the evaluation confirms that the developed tool set LiveDroid, including an Android Studio plugin and an APK patcher, is able to find the critical app state from a large space of candidate access paths and substantially boost reliability of apps running in volatile runtime environments through code generation.

Finally, this dissertation proposes static user transaction graph (SUTG), which captures critical temporal and causality relations among concurrently executed callbacks during the user transactions. SUTG can be automatically constructed by static analysis and be leveraged as the basis for various mobile instrumentation and testing scenarios. In this work, we applied SUTG in two applications: automatically profiling user-perceived latency and managing idle resources for app state testing. The SUTG-based latency profiler requires no modifications to the underlying mobile OS, and can help developers find critical execution paths in user transactions. Both applications can free developers from tedious and error-prone manual instrumentation tasks. Finally, our evaluation using a wide range of real-world mobile apps from Google Play and F-Droid show that the SUTG-based tools can be successfully applied to real-world apps, provide accurate measurements, and requires little or no interventions from developers.

Bibliography

[1] Americans check their phones 80 times a day: study. `https://nypost.com/2017/11/08/americans-check-their-phones-80-times-a-day-study/`. Accessed: 2020-06-30.

[2] How smartphones and mobile internet have changed our lives. `https://saucelabs.com/blog/how-smartphones-and-mobile-internet-have-changed-our-lives`. Accessed: 2020-07-01.

[3] Input events overview. `https://developer.android.com/guide/topics/ui/ui-events`. Accessed: 2021-03-05.

[4] Low ram configuration. `https://source.android.com/devices/tech/perf/low-ram`. Accessed: 2020-07-30.

[5] Mobile marketing statistics compilation. `https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/`. Accessed: 2020-06-30.

[6] Sensorevent. `https://developer.android.com/reference/android/hardware/SensorEvent`. Accessed: 2021-03-05.

[7] Signing Your Applications. `http://developer.android.com/tools/publishing/app-signing.html`. Accessed: 2020-07-22.

[8] Barcode scanner. `https://play.google.com/store/apps/details?id=com.google.zxing.client.android`, 2018. Accessed: 2018-04-22.

[9] The Google I/O 2017 Android app. `https://github.com/google/iosched`, 2018. Accessed: 2018-04-22.

[10] Loop - habit tracker. `https://play.google.com/store/apps/details?id=org.isoron.uhabits`, 2018. Accessed: 2018-04-22.

[11] Material design file manager for Android. `https://github.com/TeamAmaze/AmazeFileManager`, 2018. Accessed: 2018-04-22.

[12] Material design music player. https://github.com/naman14/Timber, 2018. Accessed: 2018-04-22.

[13] Telegram for Android source. https://github.com/DrKLO/Telegram, 2018. Accessed: 2018-04-22.

[14] View the java heap and memory allocations with memory profiler. https://developer.android.com/studio/profile/memory-profiler, 2018. Accessed: 2018-04-29.

[15] Wifianalyzer. https://play.google.com/store/apps/details?id=com.vrem.wifianalyzer, 2018. Accessed: 2018-04-22.

[16] apksigner - APK Signing and Verification Tool. https://developer.android.com/studio/command-line/apksigner, 2019. Accessed: 2020-07-22.

[17] jarsigner - JAR Signing and Verification Tool. http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html, 2019. Accessed: 2020-07-22.

[18] Resource types. https://developer.android.com/guide/topics/resources/available-resources.html, 2019. Accessed: 2020-07-12.

[19] Smartphone market share. http://www.idc.com/promo/smartphone-market-share/, 2019. Accessed: 2020-07-01.

[20] A tool for reverse engineering android apk files. http://ibotpeaches.github.io/Apktool/, 2019. Accessed: 2020-07-22.

[21] Alarm klock. https://play.google.com/store/apps/details?id=com.angrydoughnuts.android.alarmclock, 2020. Accessed: 2020-07-22.

[22] Android bundle. https://developer.android.com/reference/android/os/Bundle.html, 2020. Accessed: 2020-12-01.

[23] Android sharedpreferences. https://developer.android.com/reference/android/content/SharedPreferences.html, 2020. Accessed: 2020-07-22.

[24] Android sharedpreferences editor. https://developer.android.com/reference/android/content/SharedPreferences.Editor.html, 2020. Accessed: 2020-07-01.

[25] Android sqlite database. https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html, 2020. Accessed: 2020-07-01.

[26] Apkpure.com. https://apkpure.com/, 2020. Accessed: 2020-07-22.

[27] Cloc: Count lines of code. https://github.com/AlDanial/cloc, 2020. Accessed: 2020-07-22.

[28] Diary - personal diary or journal for android. https://github.com/billthefarmer/diary, 2020. Accessed: 2020-07-22.

[29] Dot file. https://en.wikipedia.org/wiki/DOT_(graph_description_language), 2020. Accessed: 2020-09-08.

[30] Glucosio for android. https://github.com/Glucosio/glucosio-android, 2020. Accessed: 2020-07-22.

[31] Imapnote 2. https://github.com/nbenm/ImapNote2, 2020. Accessed: 2020-07-22.

[32] K-9 mail. https://github.com/k9mail/k-9, 2020. Accessed: 2020-07-22.

[33] Mapbox. https://github.com/mapbox/mapbox-gl-native, 2020. Accessed: 2020-09-08.

[34] Multi-window support. https://developer.android.com/guide/topics/ui/multi-window.html, 2020. Accessed: 2020-07-12.

[35] Open mf. https://github.com/openMF/android-client, 2020. Accessed: 2020-09-08.

[36] Stackoveflow question 456211. https://stackoverflow.com/questions/456211/activity-restart-on-rotation-android, 2020. Accessed: 2020-09-08.

[37] Supporting different languages and cultures. https://developer.android.com/training/basics/supporting-devices/languages.html, 2020. Accessed: 2020-07-12.

[38] Tileview library. https://github.com/moagrius/TileView, 2020. Accessed: 2020-09-08.

[39] 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, 2020. Accessed: 2020-07-12.

[40] Vlille checker. https://play.google.com/store/apps/details?id=com.vlille.checker, 2020. Accessed: 2020-07-22.

[41] Weather & clock widget for android. https://play.google.com/store/apps/details?id=com.devexpert.weather, 2020. Accessed: 2020-07-22.

[42] Wittr. https://github.com/scoute-dich/Weather, 2020. Accessed: 2020-07-22.

[43] Wordpress-android. https://github.com/wordpress-mobile/WordPress-Android, 2020. Accessed: 2020-09-08.

[44] zipalign: an archive alignment tool. https://developer.android.com/studio/command-line/zipalign.html, 2020. Accessed: 2020-07-01.

[45] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 83–93. ACM, 2015.

[46] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles techniques and tools. 2007. *Google Scholar Google Scholar Digital Library Digital Library*, 2006.

[47] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa Gil. Live heap space analysis for languages with garbage collection. In *Proceedings of the 2009 international symposium on Memory management*, pages 129–138, 2009.

[48] alliedmarketresearch. Mobile application market by marketplace. `https://www.alliedmarketresearch.com/mobile-application-market`, 2020. Accessed: 2020-04-30.

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

[50] Android. Keeping your app responsive. `https://developer.android.com/training/articles/perf-anr`, 2020. Accessed: 2020-04-30.

[51] Android. Android activity. `https://developer.android.com/reference/android/app/Activity`, 2021. Accessed: 2021-03-05.

[52] Android. Android asynctask. `https://developer.android.com/reference/android/os/AsyncTask`, 2021. Accessed: 2021-03-05.

[53] Android. Android broadcasts. `https://developer.android.com/guide/components/broadcasts`, 2021. Accessed: 2021-03-05.

[54] Android. Android contentprovider. `https://developer.android.com/reference/android/content/ContentProvider`, 2021. Accessed: 2021-03-05.

[55] Android. Android espresso. `https://developer.android.com/training/testing/espresso`, 2021. Accessed: 2021-03-05.

[56] Android. Android jetpack. `https://developer.android.com/jetpack`, 2021. Accessed: 2021-03-05.

[57] Android. Android nativeactivity. `https://developer.android.com/reference/android/app/NativeActivity`, 2021. Accessed: 2021-03-05.

[58] Android. App crawler. `https://developer.android.com/training/testing/crawler`, 2021. Accessed: 2021-03-05.

[59] Android. Create ui tests with espresso test recorder. `https://developer.android.com/studio/test/espresso-test-recorder`, 2021. Accessed: 2021-03-05.

[60] Android. dumpsys – android command line tool. `https://developer.android.com/studio/command-line/dumpsys`, 2021. Accessed: 2021-03-05.

[61] Android. Espresso idling resources. https://developer.android.com/training/testing/espresso/idling-resource, 2021. Accessed: 2021-03-05.

[62] Android. Firefox focus: The companion browser. https://github.com/mozilla-mobile/focus-android/, 2021. Accessed: 2021-03-05.

[63] Android. Intentservice. https://developer.android.com/reference/android/app/IntentService, 2021. Accessed: 2021-03-05.

[64] Android. Measure app performance with android profiler. https://developer.android.com/studio/profile/android-profiler, 2021. Accessed: 2021-03-05.

[65] Android. Overview of memory management. https://developer.android.com/topic/performance/memory-overview, 2021. Accessed: 2021-03-05.

[66] Android. Services overview. https://developer.android.com/guide/components/services, 2021. Accessed: 2021-03-05.

[67] Android. Ui automator. https://developer.android.com/training/testing/ui-automator, 2021. Accessed: 2021-03-05.

[68] Android. Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey, 2021. Accessed: 2021-03-05.

[69] Android. Viewmodel overview. https://developer.android.com/topic/libraries/architecture/viewmodel, 2021. Accessed: 2021-03-05.

[70] Apple. About the virtual memory system. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html, 2020. Accessed: 2020-09-08.

[71] Apple. Preserving your app's ui across launches. https://developer.apple.com/documentation/uikit/view_controllers/preserving_your_app_s_ui_across_launches, 2020. Accessed: 2020-10-06.

[72] Apple. Preserving your app's ui across launches. https://developer.apple.com/documentation/uikit/view_controllers/preserving_your_app_s_ui_across_launches, 2020. Accessed: 2020-09-08.

[73] Apple. Reducing disk writes. https://developer.apple.com/documentation/xcode/improving_your_app_s_performance/reducing_disk_writes, 2020. Accessed: 2020-09-08.

[74] Apple. Reducing your app's memory use. https://developer.apple.com/documentation/xcode/improving_your_app_s_performance/reducing_your_app_s_memory_use, 2020. Accessed: 2020-09-08.

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

[76] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[77] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. Detecting behavior anomalies in graphical user interfaces. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 201–203, 2017.

[78] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 641–660, 2013.

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

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

[81] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[82] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8. ACM, 2012.

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

[84] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, volume 88 of *PLDI '88*, pages 47–56, New York, NY, USA, 1988. ACM.

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

[86] Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International journal of human-computer interaction*, 17(3):333–356, 2004.

[87] Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 665–687. Springer, 2012.

[88] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. 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*, pages 779–790. ACM, 2014.

[89] F-Droid. F-droid. `https://f-droid.org/`, 2020. Accessed: 2020-07-22.

[90] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. Efficiently manifesting asynchronous programming errors in android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 486–497, New York, NY, USA, 2018. Association for Computing Machinery.

[91] Umar Farooq and Zhijia Zhao. Runtimedroid: Restarting-free runtime change handling for android apps. In *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 110–122, New York, NY, USA, 2018. ACM.

[92] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. Livedroid: Identifying and preserving mobile app state in volatile runtime environments. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[93] George Fourtounis and Yannis Smaragdakis. Deep static modeling of invokedynamic. In *33rd European Conference on Object-Oriented Programming*, 2019.

[94] GitHub. Github. `https://github.com/`, 2020. Accessed: 2020-09-10.

[95] Google. Android developer guides: Handle configuration changes. `https://developer.android.com/guide/topics/resources/runtime-changes`, 2020. Accessed: 2020-04-30.

[96] Google. Android processes and application lifecycle. `https://developer.android.com/guide/components/activities/process-lifecycle`, 2020. Accessed: 2020-04-30.

[97] Google. Android saving ui states. `https://developer.android.com/topic/libraries/architecture/saving-states`, 2020. Accessed: 2020-04-30.

[98] Google. A Java serialization/deserialization library to convert Java objects into JSON and back. `https://github.com/google/gson`, 2020. Accessed: 2020-4-30.

[99] GooglePlay. Google play. `https://play.google.com/`, 2020. Accessed: 2020-07-22.

[100] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 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*, page 11. ACM, 2010.

[101] Shuai Hao, Ding Li, William G.J. Halfond, and Ramesh Govindan. Sif: A selective instrumentation framework for mobile applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, page 167–180, New York, NY, USA, 2013. Association for Computing Machinery.

[102] Heros. Heros ifds/ide solver. `https://github.com/Sable/heros/`, 2020. Accessed: 2020-07-22.

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

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

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

[106] Gang Huang, Mengwei Xu, Felix Xiaozhu Lin, Yunxin Liu, Yun Ma, Saumay Pushp, and Xuanzhe Liu. Shuffledog: Characterizing and adapting user-perceived latency of android apps. *IEEE Transactions on Mobile Computing*, 16(10):2913–2926, 2017.

[107] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, page 244–256, New York, NY, USA, 1979. Association for Computing Machinery.

[108] Yu Kang, Yangfan Zhou, Min Gao, Yixia Sun, and Michael R. Lyu. Experience report: Detecting poor-responsive ui in android applications. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 490–501, 2016.

[109] Yu Kang, Yangfan Zhou, Hui Xu, and Michael R. Lyu. Diagdroid: Android performance diagnosis via anatomizing asynchronous executions. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 410–421, New York, NY, USA, 2016. Association for Computing Machinery.

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

[111] David Kavaler, Daryl Posnett, Clint Gibler, Hao Chen, Premkumar T Devanbu, and Vladimir Filkov. Using and asking: Apis used in the android market and asked about in stackoverflow. In *SocInfo*, pages 405–418. Springer, 2013.

[112] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[113] Duling Lai and Julia Rubin. Goal-driven exploration for android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 115–127, 2019.

[114] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887. USENIX Association, July 2020.

[115] Seokjun Lee and Hojung Cha. User interface-level qoe analysis for android application tuning. *Pervasive Mob. Comput.*, 40(C):382–396, September 2017.

[116] Seokjun Lee, Chanmin Yoon, and Hojung Cha. User interaction-based profiling system for android application tuning. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, page 289–299, New York, NY, USA, 2014. Association for Computing Machinery.

[117] K Rustan M Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–515. Springer, 2004.

[118] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[119] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[120] 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. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 519–530. ACM, 2014.

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

[122] Yu Lin and Danny Dig. Check-then-act misuse of java concurrent collections. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 164–173. IEEE, 2013.

[123] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 224–235. IEEE, 2015.

[124] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 341–352. ACM, 2014.

[125] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[126] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1013–1024, New York, NY, USA, 2014. Association for Computing Machinery.

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

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

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

[130] Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan, and Jian Zhang. Static asynchronous component misuse detection for android applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 952–963, New York, NY, USA, 2020. Association for Computing Machinery.

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

[132] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 267–280. ACM, 2012.

[133] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 107–120, 2012.

[134] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[135] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. CGO '14, page 143–153, New York, NY, USA, 2014. Association for Computing Machinery.

[136] Alireza Sahami Shirazi, Niels Henze, Tilman Dingler, Kai Kunze, and Albrecht Schmidt. 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*, pages 362–371. ACM, 2013.

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

[138] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. *ACM SIGPLAN Notices*, 50(6):175–185, 2015.

[139] Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. Directed synthesis of failing concurrent executions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 430–446, 2016.

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

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

[142] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 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*, pages 864–880. ACM, 2016.

[143] ADB Shell. Adb shell commands. `http://developer.android.com/tools/help/shell.html`, 2020. Accessed: 2020-07-22.

[144] Soot. Soot: a java optimization framework. `https://www.sable.mcgill.ca/soot/`, 2020. Accessed: 2020-07-01.

[145] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

[146] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

[147] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *SIGPLAN Not.*, 40(10):59–76, October 2005.

[148] StackOverflow. Stackoverflow. `https://stackoverflow.com/`, 2020. Accessed: 2020-07-22.

[149] Statista. Mobile operating systems' market share worldwide from january 2012 to july 2020. `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`, 2020. Accessed: 2020-09-08.

[150] statista. Number of smartphone users worldwide 2016-2021. `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`, 2020. Accessed: 2021-04-30.

[151] statista. Worldwide mobile app revenues in 2014 to 2023. `https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/`, 2021. Accessed: 2020-04-28.

[152] Ting Su. Fsmdroid: Guided gui testing of android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 689–691, 2016.

[153] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 245–256, New York, NY, USA, 2017. Association for Computing Machinery.

[154] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 210–225. Springer International Publishing, 2013.

[155] Yan Wang and Atanas Rountev. Profiling the responsiveness of android applications via automated resource amplification. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, page 48–58, New York, NY, USA, 2016. Association for Computing Machinery.

[156] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, page 137–148, New York, NY, USA, 2012. Association for Computing Machinery.

[157] Diyu Wu, Dongjie He, Shiping Chen, and Jingling Xue. Exposing android event-based races by selective branch instrumentation. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 265–276, 2020.

[158] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*, pages 98–122. Springer, 2009.

[159] Lei Xue, Chenxiong Qian, and Xiapu Luo. Androidperf: A cross-layer profiling system for android applications. In *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, pages 115–124, 2015.

[160] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. ISSTA '11, page 155–165, New York, NY, USA, 2011. Association for Computing Machinery.

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

[162] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. 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*, pages 183–192. IEEE, 2014.

[163] Lide Zhang, David R Bild, Robert P Dick, Z Morley Mao, and Peter Dinda. Panappticon: Event-based tracing to measure mobile application and platform performance. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2013.

[164] Wenhua Zhao, Zhenkai Ding, Mingyuan Xia, and Zhengwei Qi. Systematically testing and diagnosing responsiveness for android apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–453, 2019.