LiveDroid: Identifying and Preserving Mobile App State in Volatile Runtime Environments

UMAR FAROOQ, University of California, Riverside, USA
ZHIJIA ZHAO, University of California, Riverside, USA
MANU SRIDHARAN, University of California, Riverside, USA
IULIAN NEAMTIU, New Jersey Institute of Technology, USA

Mobile operating systems, especially Android, expose apps to a volatile runtime environment. The app state that reflects past user interaction and system environment updates (e.g., battery status changes) can be destroyed implicitly, in response to runtime configuration changes (e.g., screen rotations) or memory pressure. Developers are therefore responsible for identifying app state affected by volatility and preserving it across app lifecycles. When handled inappropriately, the app may lose state or end up in an inconsistent state after a runtime configuration change or when users return to the app.

To free developers from this tedious and error-prone task, we propose 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.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Software reliability*; • **Human-centered computing → Ubiquitous and mobile computing**.

Additional Key Words and Phrases: Runtime Configuration Change, Software Restart, Static Analysis, Android

ACM Reference Format:
Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. LiveDroid: Identifying and Preserving Mobile App State in Volatile Runtime Environments. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 160 (November 2020), 30 pages. https://doi.org/10.1145/3428228

1 INTRODUCTION

Smartphones are in wide use (there were 3.2 billion smartphone users worldwide in 2019 [statista 2020]) and mobile apps have a substantial economic impact (the mobile app market is projected to reach $407 billion by 2026 [alliedmarketresearch 2020]). Hence, there is an impetus for ensuring and improving mobile app reliability. Building reliable mobile apps poses additional complications

Authors' addresses: Umar Farooq, University of California, Riverside, USA, ufaro001@ucr.edu; Zhijia Zhao, University of California, Riverside, USA, zhijia@cs.ucr.edu; Manu Sridharan, University of California, Riverside, USA, manu@cs.ucr.edu; Iulian Neamtiu, New Jersey Institute of Technology, USA, ineamtiu@njit.edu.

2475-1421/2020/11-ART160
https://doi.org/10.1145/3428228

Table 1. Example State Issues Triggered by Volatile Runtime Environments.

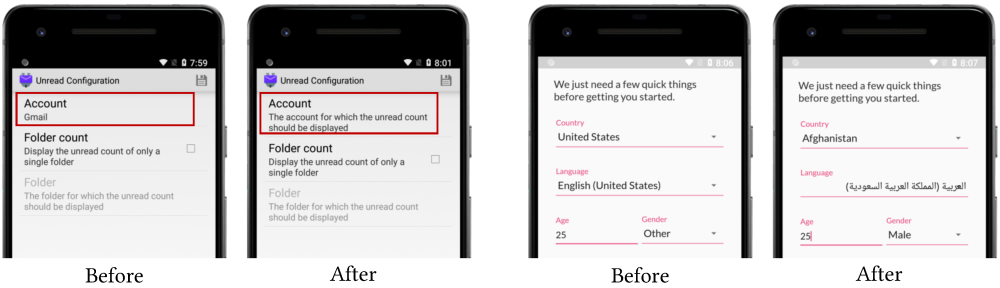
GitHub Repo	Issue ID	Issue Description
WordPress [wor 2020]	12223	Rotate the device (either from portrait to landscape or vice versa) while the tag editor is open in Post Settings, the tags are removed.
K9 Mail [k9 2020]	4519	Open *General settings*, click on the search icon, enter some text, then change screen orientation, the app crashes.
K9 Mail [k9 2020]	4936	Create *Unread widget*, click on *Account* and select user mail-box, then rotate the device, the selected account is lost.
OpenMF [ope 2020]	829	Run the app, generate a collection sheet, select office from the drop-down, change the orientation. The state gets refreshed.
TileView [til 2020]	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 [map 2020]	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 [glu 2020]	431	Open the app, fill user information such as language, gender and age, then change the orientation by rotating, the user inputs are lost.

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 ↔ 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) [Google 2020b]. 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 [Apple 2020d; Google 2020h]. This is due to mobile OSes, including both iOS and Android, eschewing swapping (i.e., paging out) [Apple 2020a; Google 2020h], to minimize flash memory wear [Apple 2020c]. 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 [Apple 2020b; Google 2020d]. 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 [Google 2020e]. 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 complex lifecycle stage transitions.



Before

7:59

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

After

8:01

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

(a) Lost account binding in K-9 Mail [k9 2020]

Before

8:06

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

Country: United States

Language: English (United States)

Age: 25

Gender: Other

After

8:07

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

Country: Afghanistan

Language: العربية (المملكة العربية السعودية)

Age: 25

Gender: Male

(b) Lost user inputs in Glucosio [glu 2020]

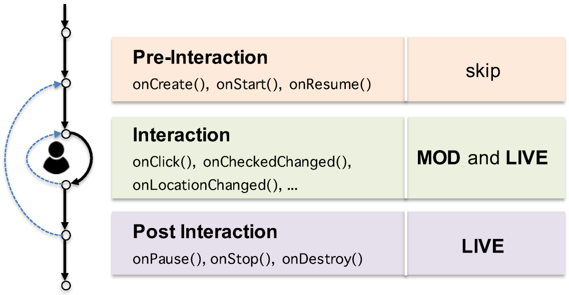
Fig. 1. Example State Issues of Two Popular Android Apps.

For example, a StackOverFlow question [sta 2020] on how to deal with initialization in the presence of activity restarting received 1394 thumbs-up.[1] Recent studies [Farooq and Zhao 2018; Shan et al. 2016] 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 1 lists a few example state issues found in several very popular GitHub repositories, including K-9 Mail [k9 2020], MapBox [map 2020], WordPress [wor 2020], TileView [til 2020], OpenMF [ope 2020], and Glucosio [glu 2020]. 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 1-(a); Glucosio loses user inputs, such as country, language, and gender, as shown in Figure 1-(b). Such unexpected app behavior negatively affects user experience.

To mitigate the aforementioned challenges, prior efforts have focused on either detecting [Shan et al. 2016] or preventing [Farooq and Zhao 2018] state inconsistency. Shan et al. [2016] 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, Farooq and Zhao [2018]'s approach is to prevent the activity from restarting at 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 [Statista 2020]) 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

[1] As of September 9th, 2020.


Pre-Interaction
onCreate(), onStart(), onResume()
skip
Interaction
onClick(), onCheckedChanged(),
onLocationChanged(), ...
MOD and LIVE
Post Interaction
onPause(), onStop(), onDestroy()
LIVE

(a) lifecycle (b) callback modeling (c) static analysis

Fig. 2. Callback Modeling and Static Analysis.

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 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

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

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 [Soot 2020]. 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 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 [F-Droid 2020], Google Play [GooglePlay 2020], 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 [Shan et al. 2016] 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.
- 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.

Next, we provide the background of this work.

2 BACKGROUND

In this section we first introduce the programming model for Android apps; then discuss the main state loss causes – activity restart and app relaunch; finally, we discuss the basic Android strategies for handling data save/restore and their inadequacy.

2.1 App Programming Model

Following the Android programming model, apps are mainly organized as *activities*, where an activity represents an individual screen that users interact with.

Activity and Its Lifecycle. Typically, an activity consists of a layout file (in XML) that specifies the GUI elements (e.g., `EditText` and `Button`) on the screen and a Java class that implements the user interaction logic behind the screen. The *lifecycle* of an activity instance is composed of a sequence of stages, including *created*, *started*, *resumed*, *paused*, *stopped*, and *destroyed* [Google 2020d], as illustrated in Figure 3. Once the activity is *resumed*, it becomes available for user interaction and accepting system updates. During the transition from one stage to the next, Android invokes its corresponding lifecycle callback(s) for developers to overwrite in order to respond to the transition.

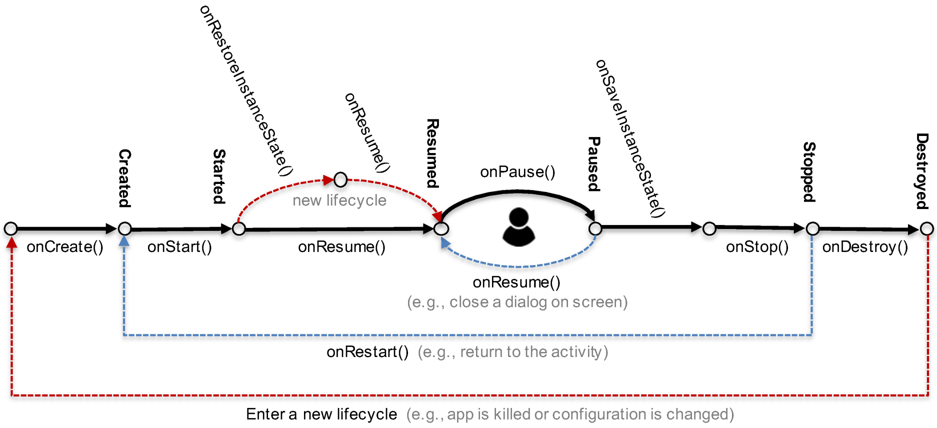


Fig. 3. Activity Lifecycle.

Note that there are a couple of "internal cycles" within the lifecycle of an activity, illustrated with the blue dashed lines in Figure 3. For example, when a pop-out dialog shows up, the activity becomes *paused*. After the user closes it, the activity comes back to *resumed*. Similarly, when the user moves away from the activity (e.g., by clicking the "Home" button), the activity goes to *stopped* and comes back to *created* once the user returns to the activity. However, once the activity is *destroyed*, for example, in response to a runtime configuration change or low-memory system killing, its current lifecycle ends. A subsequent access to the activity (after the configuration change or system relaunching) would correspond to a new activity instance, illustrated with the red dashed line in Figure 3. This is the scenario where the necessary instance state of the activity needs to be preserved. The routines used for saving and restoring are also illustrated in Figure 3, namely `onSaveInstanceState()` and `onRestoreInstanceState()`, for which we will provide more details later.

Events and Callbacks. Android apps treat both user actions and system status changes (e.g., a location change) in a unified manner, as *events*. By implementing and registering event handlers, developers make the app respond to specific events. For example, when the user clicks a button or the phone location has changed, the corresponding event handlers `onClick()` or `onLocationChanged()` will be automatically invoked. Following the convention, we refer to these event handlers as *callbacks*. Depending on whether the event is triggered by the system (e.g., location changes) or the user (e.g., clicks), the callbacks can be grouped into *system callbacks* and *UI callbacks*, respectively.

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*. 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 3). During the restart, a fresh activity instance is created and used for the subsequent user interaction. This process is known as *runtime change handling* [Google 2020b]. Besides screen size changes, other runtime

Table 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

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 [Farooq and Zhao 2018], 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 [Google 2020h]. When the memory pressure becomes high enough, the system will start to kill background apps by terminating their underlying Linux processes to reclaim memory [Google 2020d]. 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 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, such that, from the user's perspective, it appears like no restarting or relaunching has ever happened.

2.3 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 3) 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 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`[Google 2020j] 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

Table 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

`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 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 [Farooq and Zhao 2018], 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 [Farooq and Zhao 2018]. In our evaluation (Section 7), 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.

3 OVERVIEW

Figure 4 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 Section 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

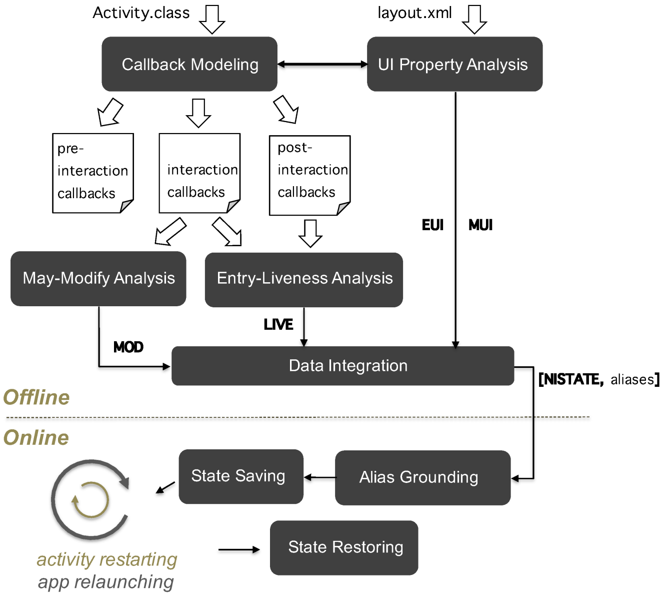


Fig. 4. Overview of LiveDroid.

"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/relaunching 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 5.

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

4 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.

4.1 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 4.1. Pre-interaction callbacks invoked before the activity becomes available for user and system interaction (i.e., *resumed*), including `onCreate()`, `onStart()`, and `onResume()`.

Definition 4.2. 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 4.3. Post-interaction callbacks executed when the activity is no longer available for user and system interaction, including `onPause()`, `onStop()`, and `onDestroy()`.

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` [Google 2020a]), 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

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

```
1  class FooActivity extends Activity{
2    ...
3    void onClick() {
4      this.f = this.a.f; //LIVE = {this.a.f.b, this.tView, this.d, this.dView}
5      if(this.f.b == true) //LIVE = {this.f.b, this.tView, this.d, this.dView}
6        this.d = this.d * 9 / 5 + 32; //LIVE = {this.tView, this.d, this.dView}
7      this.t = Calendar.getInstance().getTime(); //LIVE = {this.tView, this.d, this.dView}
8      updateViews(this.t, this.d); //LIVE = {this.t, this.tView, this.d, this.dView}
9      //LIVE = { }
10   }
11   void updateViews(Time t, Degree d) {
12     this.tView.setText(t); //LIVE = {t, this.tView, d, this.dView}
13     this.dView.setText(d); //LIVE = {d, this.dView}
14   }
15 }
```

Code 1. Example of liveness analysis.

in Section 5). 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 5). Based on the callback modeling, the analyses will become more focused, as we will show next.

4.2 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 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 [Aho et al. 2006], garbage collection [Albert et al. 2009], 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 4.4. 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 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* [Aho et al. 2006]. Initially, at the exit of a callback, no access path is live (see line 9 in Code 1). As the analysis traverses backwards, depending on whether the statement is a reference copy statement (like the one in

[4] Pre-interaction callbacks execute during a restart before instance state is restored (`onRestoreInstanceState()` in Figure 3), hence they cannot rely on saved instance state.

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` 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 1.

$$\begin{cases} LIVEOUT[i] = \bigcup_{s \in succ[i]} LIVEIN[s] \\ LIVEIN[i] = \begin{cases} LIVEOUT[i].\text{prefixsub}(LHS[i], RHS[i]) & \text{if } i \text{ is a ref copy} \\ USE[i] \cup (LIVEOUT[i] - DEF[i]) & \text{otherwise} \end{cases} \end{cases} \tag{1}$$

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 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 May-Modify Analysis

According to the modification property (mentioned at the beginning of Section 4), 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.

Code 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 2). Going backwards, for each statement that modifies a field, the analysis adds the statement’s *DEF* access path to the current

```
1  class FooActivity extends Activity{
2    ...
3    void onClick() {
4      if(mCheckBox.isSet() == true) { //MOD = {this.pView, this.u.name, this.p}
5        this.u.name = mEditText.getText(); //MOD = {this.pView, this.u.name, this.p}
6        this.p = this.u; //MOD = {this.pView, this.p}
7        this.pView.setText(this.p.name); //MOD = {this.pView}
8      }
9      //MOD = { }
10   }
11 }
```

Code 2. Example of may-modify analysis.

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 2 formally captures this analysis.

$$\begin{cases} MODOUT[i] = \bigcup_{s \in succ[i]} MODIN[s] \\ MODIN[i] = \begin{cases} MODOUT[i].\text{prefixsub}(DEF[i], RHS[i]) & \text{if } DEF[i] \text{ is a local var} \\ MODOUT[i] \cup DEF[i] & \text{if } DEF[i] \text{ is a field} \end{cases} \end{cases} \tag{2}$$

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.

4.4 UI Property Analysis

As mentioned in Section 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*.

4.5 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.

$$\begin{cases} NISTATE_{in} = MOD \cap_{alias,field} LIVE \\ NISTATE_{ex} = EUI \cup MUI \\ NISTATE = NISTATE_{in} \cup NISTATE_{ex} \\ [NISTATE, aliases] = \text{aliasing}(NISTATE, LIVE) \end{cases} \tag{3}$$

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

Complexities in Data Integration. For the external state, the union operation $EUI \cup MUI$ simply combines the two sets together. However, for the internal state, the intersection $MOD \cap LIVE$ 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` $\in MOD$ but `this.a.b` $\notin LIVE$ and `this.b` $\in LIVE$ but `this.b` $\notin MOD$. A conventional intersection $MOD \cap LIVE$ 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 $MOD \cap LIVE$. 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 [Sridharan et al. 2013] 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.

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

- Second, the intersection *MOD* ∩ *LIVE* should also be *field-sensitive* to preserve precision. For example, if `this.a` ∈ *MOD* and `this.a.b` ∈ *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 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*.

5 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.

5.1 Alias Grounding

The task of alias grounding is to verify the statically identified aliases among access paths as in $[NISTATE, aliases]$ in Equation 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.

5.2 State Saving and Recovering

As mentioned in Section 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 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

```
 1  class FooActivity extends Activity{
 2    void onSaveInstanceState(Bundle state) { ...
 3      state.putInt("int_x", this.x); //primitive
 4      state.putString("obj_b", gson.toJson(this.b)); //object
 5      if(this.a.b == this.b)
 6        state.putBoolean("a_b=b", true); //save the alias relation
 7      else //field
 8        state.putString("obj_a_b", gson.toJson(this.a.b));
 9    }
10    void onRestoreInstanceState(Bundle savedState) { ...
11      this.x = savedState.getInt("int_x");
12      String str = savedState.getString("obj_b");
13      this.b = gson.fromJson(str, B.class);
14      is_alias = savedState.getBoolean("a_b=b", false);
15      if(is_alias)
16        this.a.b = this.b;
17      else {
18        String str = savedState.getString("obj_a_b");
19        this.a.b = gson.fromJson(str, B.class);
20      }
21    }
22  }
```

Code 3. Saving and restoring internal state.

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 [Google 2020g] 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.

- ***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` ⟶ `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` ⟶ `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 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

```
1  class FooActivity extends Activity{
2    void onSaveInstanceState(Bundle state) { ...
3      TextView view = findViewById(R.id.text_time);
4      state.putString("text_time", view.getText()); //property of static view
5      state.putBoolean("tView_ref", this.tView.getViewId() == R.id.text_time); //ref
6      state.putString("dView", this.dView.getText()); //property of dynamic view
7      state.putId("dView_parent", getParentId(this.dView)); //parent GUI of dyn. view
8    }
9    void onRestoreInstanceState(Bundle savedState) { ...
10     TextView view = findViewById(R.id.text_time);
11     view.setText(savedState.getString("text_time")); //restore property
12     if(savedState.getBoolean("tView_ref", false))
13       this.tView = view; //redirect GUI reference to new GUI instance
14     if(this.dView == NULL) { //if not created during pre-interaction
15       this.dView = new TextView(this);
16       View parent = findViewById(savedState.getInt("dView_parent"));
17       parent.add(this.dView); //attach the dynamic GUI to its parent GUI
18     }
19     this.dView.setText(savedState.getString("dView")); //restore property
20   }
21 }
```

Code 4. Saving and restoring external state.

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* [Leino and Müller 2004]. 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. 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 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`[Google 2020a], `Service`[Google 2020i], and `IntentService`[Google 2020f]. 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 [Lin et al. 2014] has shown that `AsyncTask` is the source of many runtime issues and the more recent component `IntentService` is preferred.

6 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 [Soot 2020] framework, including Heros [Bodden 2012], Spark [Lhoták and Hendren 2003], and FlowDroid [Arzt et al. 2014]. The Soot [Soot 2020] framework provides an easy-to-manipulate intermediate representation (Jimple) for analyzing Java programs, and Heros [Bodden 2012] 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 [Arzt et al. 2014] 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 [Lhoták and Hendren 2003]; 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 [Bodden 2012; Reps et al. 1995], 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 [Lhoták and Hendren 2003] 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 5), 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 [Arzt et al. 2014] 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 [Fourtounis and Smaragdakis 2019] 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` [Google 2020c] or apps written in Kotlin.

7 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.

7.1 Methodology

To evaluate LiveDroid, we crawled 1,033 packages from F-Droid [F-Droid 2020] 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 [Github 2020] or 5K downloads on GooglePlay [GooglePlay 2020]. As shown in Table 5, they include some highly influential

Table 4. 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%)

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.

7.2 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 [Shan et al. 2016] 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 4 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` [Fourtounis and Smaragdakis 2019]).

Time Cost. We measured the time required for performing the static analysis on *GroupS*. The results are reported in Table 5 under Column “Time”. For most apps (30/36), the 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 5. 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

Table 5. Detailed static analysis results from applying LiveDroid-Analyzer on *GroupS*.
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 [Shan et al. 2016],
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)

App Information						Static Analysis Results							
#	Package	Stars	Popu.	ACT	MUT	EX	UIC	MOD	LIVE	IN	S	Alias	Time
1	au.com.wallaceit.reddinator	30	50K+	18	1,529	32	23	42	99	7	39	0	21
2	com.alaskalinuxuser.hourglass	5	5K+	2	25	0	0	13	15	6	6	0	2
3	com.dozingcatsoftware.asciicam	105	100K+	5	102	0	0	7	31	5	5	1(2)	3
4	com.freerdp.afreerdp	4,297	100K+	30	254	0	0	12	55	10	10	1(2)	14
5	com.fsck.k9	4,659	5M+	29	3,275	0	0	59	241	25	25	1(2)	62
6	com.github.axet.binauralbeats	16	50K+	23	549	1	2	90	252	49	50	0	40
7	com.github.xloem.qrstream	29	-	4	25	2	2	12	28	10	12	0	54
8	com.ihunda.android.binauralbeat	135	1M+	1	225	0	0	0	11	0	0	1(2)	3
9	com.kiminonawa.mydiary	1,402	-	14	2,542	0	0	8	31	8	8	0	24
10	com.llamacorp.equate	45	10K+	13	201	0	0	27	86	25	25	0	7
11	com.namelessdev.mpdroid	557	100K+	26	172	5	5	140	172	58	63	1(2)	42
12	com.ringdroid	692	-	3	142	0	0	18	30	10	10	0	7
13	com.sagar.screenshift2	47	1M+	36	297	10	10	12	45	0	10	0	11
14	com.tastycactus.timesheet	42	-	6	67	19	19	10	52	8	27	0	2
15	de.baumann.browser	387	10K+	12	319	17	12	31	61	24	41	0	11
16	de.onyxbits.listmyapps	57	100K+	5	69	10	10	5	15	4	14	0	5
17	de.schildbach.wallet	1,818	1M+	13	415	0	0	24	61	17	17	0	507
18	de.smasi.tickmate	78	1K+	10	173	13	13	36	22	10	23	0	6
19	jackpal.androidterm	2,318	10M+	21	185	0	0	53	98	34	34	0	61
20	jp.sblo.pandora.aGrep	18	10K+	21	65	0	0	2	30	1	1	0	1
21	moe.minori.pgpclipper	18	-	5	54	0	0	15	33	12	12	0	5
22	net.kervala.comicsreader	1	100K+	11	166	2	2	41	97	29	31	1(2)	9
23	nl.asymmetrics.droidshows	53	-	6	283	30	19	7	43	5	35	0	7
24	org.billthefarmer.diary	98	-	3	48	5	4	4	16	2	7	1(2)	4
25	org.billthefarmer.tuner	91	-	3	171	0	0	6	6	6	6	0	16
26	org.disrupted.rumble	138		23	553	13	10	20	48	13	26	0	24
27	org.glucosio.android	324	-	17	621	35	35	1	7	0	35	0	116
28	org.gnucash.android	987	100K+	22	1,169	18	10	3	29	2	20	1(2)	1,779
29	org.horaapps.leafpic	2,948	-	11	3,038	0	0	20	57	13	13	0	47
30	org.openintents.notepad	38	50K+	9	127	2	2	20	38	12	14	1(2)	6
31	org.secuso.privacyfriendlynotes	46	5K+	11	145	22	15	21	47	16	38	0	10
32	org.secuso...tapemeasure	8	5K+	11	644	5	5	21	43	15	20	2(4)	6
33	org.yaxim.androidclient	91	100K+	10	205	0	0	10	25	8	8	0	68
34	ru.henridellal.emerald	42	10K+	8	108	7	7	17	27	6	13	0	5
35	se.bitcraze.crazyfliecontrol2	85	10K+	12	713	7	7	55	189	34	41	2(4)	22
36	tellh.com.gitclub	620	-	15	1,102	0	0	10	84	7	7	0	49
Total				469	19,778	255	212	872	2,224	491	746	13	-

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* 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, 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 [Shan et al. 2016]. 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 [Shan et al. 2016] 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 5 and 6) with the state actually saved by developers in the original app code (Column S_s in Table 6). 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 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 7 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.

Table 6. 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

Table 7. 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

```
// 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
  ...
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_bootloader);
    ...
   this.mConsoleTextView = (TextView) findViewById(R.id.bootloader_statusLine);
  }
  @Override
  protected void onPostExecute(String result) { ... //callback of an AsyncTask
    appendConsoleError("Firmware file can not be found.");
    ...
  }
  public void appendConsoleError(String status) { ...
    this.mConsoleTextView.append("\n" + status);
    ...
  }
  public void startFlashProcess(final View view) { //a click handler callback
    this.mConsoleTextView.setText("");
    ...
  }
}
```

Code 5. Case Study: False Positive and True Positive.

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 6. 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 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.

7.3 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 8. 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 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.

Table 8. 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 9. 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

Besides time costs, code generation also increases the size of source code. Table 9 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 [Shan et al. 2016]. 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 10 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 and S in Table 5) 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), 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 7.2). The results show that all 46 issues were successfully fixed, because

Table 10. 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

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.

8 RELATED WORK

This section summarizes relevant work on mobile app reliability, especially work pertaining to app restarting. AppDoctor [Hu et al. 2014] identifies app issues related to activity restart based on orientation change events (a common restart trigger). However, AppDoctor does not reveal the root causes of issues, neither does it propose a solution to the identified issues. Zaeem and others [Zaeem et al. 2014] have built GUI models to generate test cases for mobile apps. Their tool supports richer interaction events such as orientation changes, pause-resume, kill-restart, and back-key events to trigger more app issues. Their report includes issues related to restarting. Adamsen and others [Adamsen et al. 2015] perform testing via neutral event sequences, including lifecycle events, such as pause-resume, pause-stop-restart, and pause-stop-destory-create. Our approach complements these efforts by providing an automatic solution to avoid states issues.

Farooq and Zhao [2018] have introduced a runtime sub-system, RuntimeDroid, that enables restarting-free handling for configuration changes. RuntimeDroid can update app resources automatically during a runtime change without restarting the activity. However, their work focuses on runtime configuration changes and is unable to address restarting triggered by high-memory pressure. In comparison, our work covers both restarting scenarios. Shan et al. [Shan et al. 2016] 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. [Lebeck et al. 2020] 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 [Apple 2020c]. 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.

More generally, there have been research efforts on the evaluation, validation, and refactoring of mobile apps with different focus compared to ours, such as identifying race conditions and energy bugs using dynamic analysis [Hsiao et al. 2014; Hu and Neamtiu 2011; Maiya et al. 2014], detecting network and GPS location bugs [Liang et al. 2014], as well as the uncovering of app quality related problems [Arijo et al. 2011; Berardinelli et al. 2010; Lillack et al. 2014; Liu et al. 2014; Muccini et al. 2012] and memory leaks [Yan et al. 2013]. Bavota and others[Bavota et al. 2012] studied the practices of refactoring and their consequences, e.g., the degree to which faults can be caused by refactoring. In addition to improving design quality, there is an increase in non-functional quality refactoring, such as refactoring global-state with thread-local-state [Schäfer et al. 2010], refactoring built-in locks with more flexible ones[Schafer et al. 2011], refactoring the concurrent programming constructs [Okur et al. 2014], and refactoring for energy efficiency [Sahin et al. 2014]. Unlike prior work, our work aims for a refactoring-based solution to address app state issues.

Static analyses with access paths have been seen in prior work, such as taint analysis [Arzt et al. 2014; Tripp et al. 2013], alias analysis [De and D'Souza 2012] and concurrency analysis [Blackshear et al. 2018; Samak et al. 2015, 2016]. FlowDroid [Arzt et al. 2014] uses access paths to propagate reachable objects through the program path under scrutiny. As alias analyses commonly [Sridharan et al. 2005; Xu et al. 2009; Yan et al. 2011] use *context-free language* (CFL) to express alias relations, fields are part of the language and access paths provide the abstraction of fields. De and D'Souza [2012] avoid construction of points-to graphs and compute access paths for reachable local variables and static fields at each program statement. In [Blackshear et al. 2018] authors use access paths for race detection between syntactically identical access paths, where access paths are rooted at callee functions. All of these works use access paths within their analyses specialized to their context, similarly we use access paths for reachable fields from `Activity` class at hand for analysis.

9 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.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers for the careful reading of the paper, and for many insightful comments and suggestions. This work is based upon the work supported in part by National Science Foundation Grant No. CCF-2007024, CNS-1617584, and a Google Research Award.

REFERENCES

2020. Glucosio for Android. https://github.com/Glucosio/glucosio-android. Accessed: 2020-07-22.

2020. K-9 Mail. https://github.com/k9mail/k-9. Accessed: 2020-07-22.

2020. MapBox. https://github.com/mapbox/mapbox-gl-native. Accessed: 2020-09-08.

2020. Open MF. https://github.com/openMF/android-client. Accessed: 2020-09-08.

2020. StackOveflow Question 456211. https://stackoverflow.com/questions/456211/activity-restart-on-rotation-android. Accessed: 2020-09-08.

2020. TileView Library. https://github.com/moagrius/TileView. Accessed: 2020-09-08.

2020. WordPress-Android. https://github.com/wordpress-mobile/WordPress-Android. Accessed: 2020-09-08.

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

Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2006. Compilers: Principles Techniques and Tools. 2007. *Google Scholar Google Scholar Digital Library Digital Library* (2006).

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

alliedmarketresearch. 2020. Mobile Application Market by Marketplace. https://www.alliedmarketresearch.com/mobile-application-market. Accessed: 2020-04-30.

Apple. 2020a. About the Virtual Memory System. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html. Accessed: 2020-09-08.

Apple. 2020b. Preserving Your App's UI Across Launches. https://developer.apple.com/documentation/uikit/view_controllers/preserving_your_app_s_ui_across_launches. Accessed: 2020-10-06.

Apple. 2020c. Reducing Disk Writes. https://developer.apple.com/documentation/xcode/improving_your_app_s_performance/reducing_disk_writes. Accessed: 2020-09-08.

Apple. 2020d. Reducing Your App's Memory Use. https://developer.apple.com/documentation/xcode/improving_your_app_s_performance/reducing_your_app_s_memory_use. Accessed: 2020-09-08.

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

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

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

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

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

Eric Bodden. 2012. 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*. ACM, 3–8.

Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science)*, James Noble (Ed.), Vol. 7313. Springer, 665–687. https://doi.org/10.1007/978-3-642-31057-7_29

F-Droid. 2020. F-Droid. https://f-droid.org/. Accessed: 2020-07-22.

Umar Farooq and Zhijia Zhao. 2018. RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps. In *Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '18)*. ACM, New York, NY, USA, 110–122. https://doi.org/10.1145/3210240.3210327

George Fourtounis and Yannis Smaragdakis. 2019. Deep Static Modeling of invokedynamic. In *33rd European Conference on Object-Oriented Programming*.

Github. 2020. Github. https://github.com/. Accessed: 2020-09-10.

Google. 2020a. Android AsyncTask. https://developer.android.com/reference/android/os/AsyncTask. Accessed: 2020-08-30.

Google. 2020b. Android Developer Guides: Handle Configuration Changes. https://developer.android.com/guide/topics/resources/runtime-changes. Accessed: 2020-04-30.

Google. 2020c. Android NativeActivity. https://developer.android.com/reference/android/app/NativeActivity. Accessed: 2020-08-30.

Google. 2020d. Android Processes and Application Lifecycle. https://developer.android.com/guide/components/activities/process-lifecycle. Accessed: 2020-04-30.

Google. 2020e. Android Saving UI States. https://developer.android.com/topic/libraries/architecture/saving-states. Accessed: 2020-04-30.

Google. 2020f. IntentService. https://developer.android.com/reference/android/app/IntentService. Accessed: 2020-09-10.

Google. 2020g. A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson. Accessed: 2020-4-30.

Google. 2020h. Overview of memory management. https://developer.android.com/topic/performance/memory-overview. Accessed: 2020-04-30.

Google. 2020i. Services overview. https://developer.android.com/guide/components/services. Accessed: 2020-09-10.

Google. 2020j. ViewModel Overview. https://developer.android.com/topic/libraries/architecture/viewmodel. Accessed: 2020-09-08.

GooglePlay. 2020. Google Play. https://play.google.com/. Accessed: 2020-07-22.

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

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

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

Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. 2020. End the Senseless Killing: Improving Memory Management for Mobile Operating Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 873–887. https://www.usenix.org/conference/atc20/presentation/lebeck

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

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction, 12th International Conference (LNCS)*, G. Hedin (Ed.), Vol. 2622. Springer, Warsaw, Poland, 153–169.

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

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

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

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

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

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

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

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

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

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

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

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

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

Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding resume and restart errors in Android applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 864–880.

Soot. 2020. Soot: a Java Optimization Framework. https://www.sable.mcgill.ca/soot/. Accessed: 2020-07-01.

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

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-to Analysis for Java. *SIGPLAN Not.* 40, 10 (Oct. 2005), 59–76. https://doi.org/10.1145/1103845.1094817

Statista. 2020. 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/. Accessed: 2020-09-08.

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

Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Vol. 7793. Springer International Publishing, 210–225. https://doi.org/10.1007/978-3-642-37057-1_15

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

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-Driven Context-Sensitive Alias Analysis for Java *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 155–165. https://doi.org/10.1145/2001420.2001440

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

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