Finding Resume and Restart Errors in Android Applications

Zhiyong Shan
School of Computer Science and Mathematics, University of Central Missouri, Warrensburg, MO, 64093, USA
shan@ucmo.edu

Tanzirul Azim
Department of Computer Science and Engineering, University of California, Riverside, CA, 92521, USA
mazim002@cs.ucr.edu

Iulian Neamtiu
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ, 07102, USA
ineamtiu@njit.edu

Abstract

Smartphone apps create and handle a large variety of "instance" data that has to persist across runs, such as the current navigation route, workout results, antivirus settings, or game state. Due to the nature of the smartphone platform, an app can be paused, sent into background, or killed at any time. If the instance data is not saved and restored between runs, in addition to data loss, partially-saved or corrupted data can crash the app upon resume or restart. While smartphone platforms offer API support for data-saving and data-retrieving operations, the use of this API is ad-hoc: left to the programmer, rather than enforced by the compiler. We have observed that several categories of bugs—including data loss, failure to resume/restart or resuming/restarting in the wrong state—are due to incorrect handling of instance data and are easily triggered by just pressing the 'Home' or 'Back' buttons. To help address this problem, we have constructed a tool chain for Android (the *KREfinder* static analysis and the *KREreproducer* input generator) that helps find and reproduce such incorrect handling. We have evaluated our approach by running the static analysis on 324 apps, of which 49 were further analyzed manually. Results indicate that our approach is (i) effective, as it has discovered 49 bugs, including in popular Android apps, and (ii) efficient, completing on average in 61 seconds per app. More generally, our approach helps determine whether an app saves too much or too little state.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability, Validation; D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools

General Terms Reliability, Verification

Keywords Mobile applications, Software restart, Data loss, Static analysis, Google Android, App testing

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984011

1. Introduction

The smartphone platform differs fundamentally from the desktop or server platforms: once started, desktop/server programs can expect to run "forever" as the OS will not attempt to kill them when the user switches among programs. In contrast, on smartphone OSes such as Android [7] or iOS [1], an app runs in a restricted mode once the user switches away from it, i.e., the app goes into the background – this is for good reasons such as protecting privacy and conserving resources. In fact, smartphone OSes frequently and periodically kill background apps for these reasons [1, 7, 10]. To help programmers preserve users' work, smartphone platforms offer API support for data-saving and data-retrieving operations. Hence programmers should write apps accordingly to deal with the possibility of apps being killed, saving any pertinent data and retrieving the data once the app is restarted. Unfortunately, many apps are not written this way: the use of the data save&retrieve API is ad-hoc: completely left to the programmer, rather than enforced by the compiler. As a result, data created by apps (such as workout data in a health&fitness app, scan settings in an antivirus app, alarm settings in an alarm clock app, or game state in a game app) can get lost when an app is switched away from. We name *KR data* the data that should be preserved across runs (kill-and-restart or pause-and-resume cycles). Motivated by this observation, we have investigated the consequences of incorrect KR data handling in Android apps. We found that several categories of bugs—loss of user's work, loss of app or device settings— are due to incorrect KR data handling; we call them *KR errors*. In Section 2 we present KR error examples, the Android restart model, and the programmer's KR handling responsibilities.

To help address KR errors, we have constructed a static analysis (*KREfinder*) that finds incorrect or inconsistent handling of KR data. The analysis works on off-the-shelf Android apps (APKs) without requiring access to the app's

source code. Constructing the analysis required surmounting several challenges: identifying KR data, finding those operations that change, save, or restore KR data; and expressing consistent save/restore operations for KR data in a manner that can be rigorously defined and implemented in a static analysis (Section 3). We have identified four main types of errors. The analysis produces a report containing potential bugs (error location and path exposing the error). To help reproduce and fix the errors, we have constructed the *KREreproducer*, which takes *KREfinder*'s report as input and generates a sequence of inputs to guide the app to the error point.

In Section 4 we describe our implementation; the implementation and datasets are available as open-source.[1] *KREfinder* extends the Flowdroid static analysis infrastructure [12] with support for tracking data and control flow between arbitrary instructions, and built a KR error finder that employs data flow and bidirectional control flow analysis (Section 4.1). To help users reproduce the errors reported by our static analysis, we have also constructed *KREreproducer*, an automated approach that, given a report, produces a sequence of input events that lead to the app state where the error manifests; from that point, the developer can kill-and-restart (or pause-and-resume) the app to reproduce the error (Section 4.2).

We provide an evaluation of our approach in Section 5. We ran our analysis on 324 apps, and chose 49 of them for further manual analysis. The apps cover a wide range of categories (from utilities to games to news), various sizes and various levels of popularity. Experiments show that our approach is: (1) effective at finding potential KR data errors, even in large and widely-popular apps, and with a low false-positive rate; (2) effective at helping users investigate and reproduce errors; (3) efficient, able to complete in 61 seconds per app on average.

In summary, our contributions in this paper are:

1. Revealing and defining *KR errors*, a new class of errors germane to smartphone apps.
2. A definition of KR data, i.e., data that should be saved and restored across resume-and-restart cycles.
3. *KREfinder*, a static analysis for identifying KR data and incorrect handling thereof.
4. *KREreproducer*, an input generator to facilitate reproducing errors.
5. An evaluation on a variety of Android apps, 324 in total; 49 apps using automatic as well as manual analysis, and 291 using automatic analysis. The evaluation has revealed 49 bugs in 37 apps, including 24 bugs in 18 very popular apps (more than 100,000 installs).

[1] `http://spruce.cs.ucr.edu/kre/`

2. Motivation

To motivate the importance of finding KR errors, we begin with some examples of such errors. Next, we discuss restarts and platform support for KR data handling in Android. Then, we show a concrete example of incorrect KR data handling and loss in the DateSlider open source app.

2.1 Examples of KR Errors

KR errors manifest in various ways. Consider Personal Work Recorder, an app for recording workout timings. If the app is restarted while a workout recording is in progress, the current workout timing information is lost, which is directly against the purpose of a workout-tracking app. The Zirco browser can lose a previously-set bookmark upon restart. The OpenSudoku and Scrambled Net games can lose their state.

Alarm Clock Plus, an alarm clock manager, upon restart loses the alarm that has been set, which is particularly undesirable when using the alarm app for an important event. The Dr.WebLight antivirus loses its custom scan option setting. In another example that will confuse or irritate the user, the Audalyzer app can prompt the user to read and accept its license even though the user has previously accepted the license. Finally, device settings can be lost upon restart, from the phone's flashlight settings in Symantec Norton Snap, to speaker phone settings in SpeakerProximity, to camera setting in Motorola Camera, to Bluetooth settings in BTHF PowerSave and FoxFi. Table 9 in Section 5 contains more substantial descriptions of the KR errors in the 49 manually-examined apps.

2.2 App Lifecycle

Android has built-in support for data save/restore and notifying apps of pause-and-resume or kill-and-restart operations.

In Android, apps are structured around "activities"; an activity roughly corresponds to a separate screen in a traditional, desktop program with a GUI. Activities are created by extending the Activity class and instantiating the subclass. Activity instances follow a lifecycle [7], moving between several states: *Stopped*, *Paused*, *Resumed*, etc., as explained shortly. The Android Framework (aka AF, the main orchestrator of app execution) automatically invokes certain on$*$() callbacks when the activity cycles among these states.

The lifecycle is illustrated in Figure 1. When the activity is *Created*, the onCreate() callback is invoked; developers can add custom initialization code here. The activity is not yet visible. The activity can move to *Started* after it was created or stopped; onStart() is invoked, and the activity becomes visible. When the activity transitions to *Resumed* (the usual operating state), onResume() is called. The activity can become *Paused* when it is partially or fully covered, e.g., by a dialog box or a drop-down menu; the onPause() callback is invoked. When the activity becomes *Stopped*, e.g., due to an incoming phone call, onStop() is invoked; the activity can be restarted, e.g., after the phone call completes,

and onRestart() is invoked. When the activity is about to be *Destroyed*, e.g., because the user has pressed the 'Back' button, onDestroy() is invoked; after that, the activity is killed.

2.3 Restart Levels

When an app is paused, stopped, or killed, data loss may occur. We now proceed to define several *restart levels* in Android, based on the amount of app state that is affected by restart; each restart level has one or more associated callback methods, automatically invoked by the AF. We first discuss the levels and then describe the restarts' impact on data.

Restart levels. The levels, restart circumstances, and callbacks are summarized in Table 1.

1. *Pause activity.* A foreground activity can become partially covered (e.g., when the user swipes to activate the drop-down menu or when a dialog box opens) or the display may be turned off; in such cases the AF invokes onPause(). When the activity comes back into the foreground, the AF invokes onResume()—note the smallest, *Paused/Resumed* restart cycle in Figure 1. While "generally" a paused activity is kept in memory hence no data is lost, the app can be killed while paused if the system needs to reclaim memory [7].
2. *Stop activity.* A paused activity can be stopped for a various reasons, e.g., an incoming call, transitioning to another activity, switching to another app, the user pressing the 'Home' button. When the app is stopped, the AF invokes onStop() rendering the activity invisible. When the app is restarted, the onRestart() callback is invoked—note the medium, *Started/Stopped* restart cycle in Figure 1. Similar to paused activities, a stopped activity is generally kept in memory hence no data is lost, but it can still be killed while stopped, to reclaim memory [7].
3. *Destroy activity.* A stopped activity is destroyed—and all its data will be disposed of—when the AF invokes onDestroy(). This happens for example when the app is killed or the user presses the 'Back' button.

Interestingly, an operation that might appear lightweight – changing the phone's orientation between vertical and horizontal – is actually quite heavyweight, as the activity is destroyed and recreated, undergoing six transitions: *Resumed* → *Paused* → *Stopped* → (*destroyed*) → *Stopped* → *Paused* → *Resumed*.

2.4 KR Data Handling and Loss

App process model and killing. In Android, an app runs as a Linux process. For separation reasons, each app has its own UID, hence UID's differ across apps and PID's differ across app runs—by "app killing" we mean that the underlying Linux process is killed. When a process is killed, all the in-memory state (hence unsaved data) is lost. The killing happens swiftly, to keep the UI responsive, so the system can eschew invoking activity-stop (onStop), activity-destroy (onDestroy), or save-data (onSaveInstanceState) callbacks. Therefore, apps can lose data if developers choose to place KR saves in such methods that might never be called.

KR Data Handling. To prevent such losses, AF provides two main support mechanisms: (1) automatically saving&restoring GUI data only; (2) manually saving&restoring arbitrary data via on∗() callbacks. Table 2 lists the callbacks that are guaranteed to be invoked when the app is killed, versus callbacks that are not guaranteed; we explain these callbacks in detail next.

Automatic save&restore. A callback named onSaveInstanceState() is provided by the AF, giving developers a chance to save KR data before the activity is killed. The default implementation of onSaveInstanceState() saves GUI state, e.g., the contents of an editable text field, meaning that certain parts the GUI state are saved and restored "for free", but other parts are the user's responsibility. [2] Unfortunately, onSaveInstanceState() is *not guaranteed* to be called before an activity is paused or stopped.[3]

Manual save&restore. To prevent data losses, the Android documentation advises developers to manually save data[4] and offers an API for storing data into files, databases, and key-value sets [11]. However, the use of (let alone *correct use of*) the data save and restore API is not enforced statically, hence KR data handling is ad-hoc and prone to errors.

Hence it is natural to expect that developers use onSaveInstanceState(), onStop(), onDestroy(), or onPause(), to save state. Unfortunately, relying on these callbacks might be problematic.

First, onSaveInstanceState() is not guaranteed to be called, as mentioned priorly, and neither is onStop() [7]. Second, onStop() and onDestroy() are "killable": per the Android documentation *"after that method returns, the process hosting the activity may be killed by the system at any time without another line of its code being executed."* [7].[5] Since onStop() is killable, onDestroy() might never be invoked. Hence the only "sure bet" for developers saving the KR data is in onPause().

[2] Per the official Android documentation: "Although the default implementation of onSaveInstanceState() saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default)" [6].

[3] Idem, "There's no guarantee that onSaveInstanceState() will be called before your activity is destroyed [...] If the system calls onSaveInstanceState(), it does so before onStop() and possibly before onPause()" [6].

[4] Idem, "Most Android apps need to save data, even if only to save information about the app state during onPause() so the user's progress is not lost. Most non-trivial apps also need to save user settings, and some apps must manage large amounts of information in files and databases" [11].

[5] onPause() is killable in pre-3.0 versions of Android. However, since only 2% of Android devices run such older versions [8] we consider onPause() to be de facto non-killable. Note though that onStop() can be eschewed due to memory pressure [7].

Level	Cause	Kill Callback	Restart Callback
1: Pause activity	Activity becomes (partially) covered; Turn off screen	onPause()	onResume()
2: Stop activity	Switch to another app; Start a new activity in the same app; Receive a phone call; Press 'Home' button	onStop()	onRestart()
3: Destroy activity	Press 'Back' button; Kill app	onDestroy()	onCreate()

Table 1: Android restart levels.

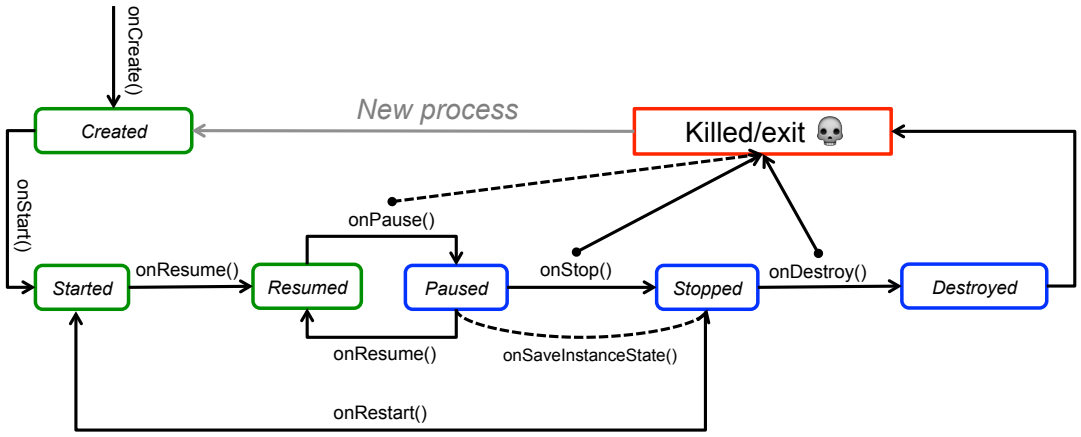


Figure 1: Android activity lifecycle (adapted from [7, 34]).

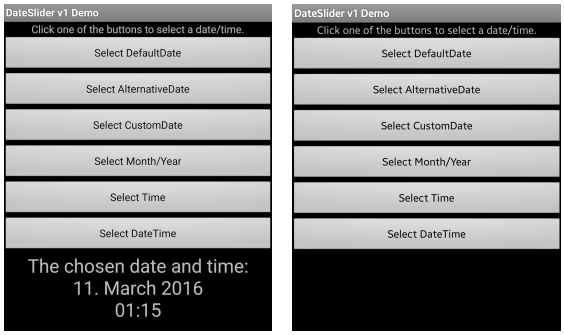
Guaranteed	*Not Guaranteed*
onPause()	onStop(), onDestroy(), onSaveInstanceState()

Table 2: Callbacks invocation guarantees upon app kill.

Third, developers might be confused whether onPause() is appropriate for saving data or not; per the Android documentation *"Generally, you should not use onPause() to store user changes (such as personal information entered into a form) to permanent storage. The only time you should persist user changes to permanent storage within onPause() is when you're certain users expect the changes to be auto-saved (such as when drafting an email)"* [9].

The caveats of over-saving the state. Note that one potential approach for avoiding state loss might be to "over-save": checkpoint and reinstate all the process state, e.g., as done in process migration [29]. However, over-saving might lead to at least two kinds of issues. First, on∗() methods are supposed to be, as Google puts it, "speedy" to avoid UI sluggishness [7]. Android might throw an "Application Not Responding" (ANR) exception when the app lingers in a callback. So an over-save might be abruptly terminated in the middle, after it has saved irrelevant fields but before it had a chance to save relevant fields, resulting in new KR errors.

Second, as with any serialization, care must be taken when saving and restoring pointers, IDs, or handles that differ across executions. An example would be a file descriptor, e.g., fileDescriptor 42 before restart would be valid, but after restart the process might not have a fileDescriptor 42, or descriptor 42 could point to another open file, etc. To address these issues a modified platform, such as OS or VM support for migration, might be necessary.



(a) Before restart **(b)** After restart

Figure 2: Dateslider KR error: the selected date, "11. March 2016", shown on the bottom, is lost upon restart.

2.5 Example: DateSlider KR Error

We now present a concrete example of a KR error in the Dateslider app, which illustrates the KR data loss upon restart and shows the source code bug responsible for it. Dateslider is a simple app that allows the user to select a date and time.

```
1  public abstract class DateSlider extends Dialog {
2    protected Calendar mTime;
3     ......
4    public void onCreate(Bundle savedInstanceState) {
5      super.onCreate(savedInstanceState);
6      if (savedInstanceState!=null) {
7        long time = savedInstanceState.getLong(
8                        "time", mTime.getTimeInMillis());
9        mTime.setTimeInMillis(time);
10     }
11       ......
12   }
13    ......
14   public void updateCalendar(Calendar calendar) {
15     mTime.setTimeInMillis(calendar.getTimeInMillis());
16   }
17    ......
18   public Bundle onSaveInstanceState() {
19     Bundle savedInstanceState =
20      super.onSaveInstanceState();
21     if (savedInstanceState==null)
22     savedInstanceState = new Bundle();
23     savedInstanceState.putLong("time",
24              mTime.getTimeInMillis());
25     return savedInstanceState;
26   }
27   protected void onDestroy() {
28    ...
29   }} ...
```

Figure 3: Source code excerpt from DateSlider.

The app contains a KR error: if the app is restarted, the user's selection is not saved.

In Figure 2 we show two screen shots: on the left, we have the app screen after the user has selected a date (*11. March 2016*); the user naturally expects that the selection will survive a resume or restart. However, due to a KR bug, when the app restarts, the selection is lost—we show this in Figure 2 on the right.

We now explain the cause of this bug. Figure 3 shows a source code excerpt from DateSlider. The activity (screen) DateSlider class has a KR field mTime. The field is updated in method updateCalendar() and saved in onSaveInstanceState(). When the app restarts, method onCreate() checks for the existence of saved data and imports it if present. Since onSaveInstanceState() is not guaranteed to be invoked (Section 2.4), the field mTime is not guaranteed to be saved, which leads to a KR error.

We now show how our analysis finds this error. First, we identify line 15 as a *KR data change*, line 23 as a *KR data save*, and the end of onDestroy(), line 29, as an *exit*. Second, we trace data-flow from the change to the save and identify line 15 (mTime.setTimeInMillis (...)) as the data change to be saved. Third, we trace control-flow from the change to exit. As onSaveInstanceState() will not be called when the app is force-stopped, there is a flow to the end of onDestroy() without saving the data change. In other words, the change in line 15 might get lost, which we report as a potential KR error. Our analysis, described next, identifies KR data that is improperly handled, which helps uncover KR errors.

3. Approach

The high-level view of our approach is presented in Figure 4: it consists of two main components, *KREfinder* and *KREreproducer*. Given an Android app (APK file) as input, *KREfinder* performs a suite of static analyses (data and control flow) to find potential KR errors, and report their location and KR error path as a summary. The summaries are in a format suitable for *KREreproducer*, that automatically generates a sequence of directed transitions (input event sequences) that land the app in the potential-error state. At this point, the app can be paused-and-resumed (or killed-and-restarted, respectively) so the error can be confirmed. The analyses consist of several phases, which we will describe shortly:

1. Identify KR data.
2. Find changes (writes) to KR data, saves/restore operations, and exits.
3. Construct the data flow from KR changes to KR saves and identify potential errors.
4. Construct the control flow from KR changes to exits and identify potential errors.
5. Generate directed transitions (input sequences) for reproducing and verifying the errors.

3.1 Identifying KR Data

Finding KR data is challenging because this data is app-specific, from time-of-day in DateSlider to workout data in Personal Work Recorder to bookmarks in Zirco Browser. By analyzing dozens of open source apps that use KR data, we gained a key insight: *KR data is a subset of fields in the application-defined classes*.

To be classified as candidate KR data, fields need to be mutable (constant or final fields are excluded, since their values cannot change) and be modified, i.e., the field must be changed, directly or through an alias, in at least one instruction. Fields that are not modified are not considered potential KR data—when app execution does not modify a field, the value of that field is the same across resume or restart cycles.

3.2 Finding KR Data Changes, Saves/Restores, and Exits

Finding KR Data Changes. With the KR data fields at hand, we proceed to finding if, and how they change. We define data changes as assignment statements where a KR field (or alias thereof) is written to. However, we ignore assignments in initialization or destruction methods, because such assignments are actually initializing or storing variables, e.g., loading data from persistent storage into fields or writing field values into persistent storage. In Android, an activity is initialized in onCreate(), onStart() and onRestart(),

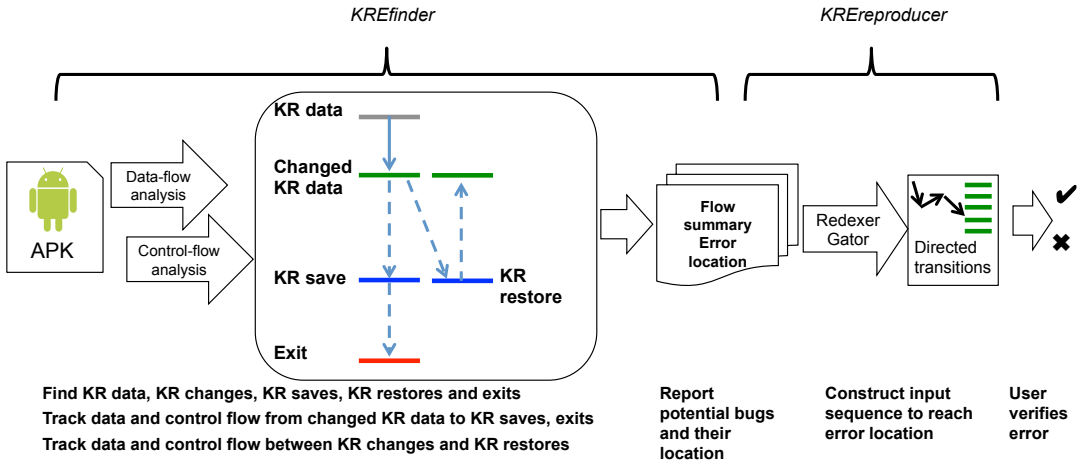


Figure 4: Overview of our approach.

and destructed in onDestroy(). Hence, if a field assignment statement appears in these callback methods, we do not identify it as a change statement. We follow method invocation transitively in the call graph: the callee methods (directly or indirectly called by these callbacks) are also treated as initialization or destruction, respectively. Finally, the construction methods of an object or class are also treated similarly. For example, init () and clinit () initialize an instance and class respectively. Once we eliminate the changes made in initialization and destruction methods, we are left with the genuine field changes that help us identify KR data.

Finding KR Data Saves and Restores. We identify data saves and restores as those call instructions to certain Android API methods that save data to persistent storage and restore data from persistent storage, respectively. However, as there are thousands of API methods in the AF, manually enumerating these methods is not feasible. We thus analyzed the Android API documentation and constructed rules to identify the API calls of interest, which are listed in Table 3 and Table 4. In the tables, $*$ matches any number of characters. For example, to find data saves, we check whether the method name starts with put when the class name is android.os.Bundle; to find data restores, we check whether the method name starts with query when the class name is android.database.sqlite.SQLiteDatabase; in other classes, we look for operations whose prefix is save or restore, respectively.

Finding Exits. The exit of an Android app or activity is different from that of a desktop application since Android apps are event-driven and do not have a main() method. We have identified three types of exit statements.

First, the end of onDestroy() method—this is the most common exit. According to the Android documentation, this callback method is the final call an activity receives before it is destroyed. This can happen either because the activity is finishing (its finish () method was invoked), or because the system is temporarily destroying this instance of the activity for cleanup or resource conservation purposes. Second, a java.lang.System.exit() call; this call terminates the currently-running Java Virtual Machine. Third, the end of onStop(). This method is called when the activity is no longer visible to the user (going from foreground to the background), because another activity is resuming, hence forcing the current foreground activity into the background. Note that an activity can be terminated without calling onDestroy() if the user force-stops the app from the app management interface. Fourth, our implementation has the option to set the end of onPause() as an exit; we have not used this option since exiting directly after onPause() is rare, as explained in Section 2.4.

Class	Method
Android.content.SharedPreferences	put∗
Android.content.SharedPreferences$Editor	put∗
Android.os.Bundle	put∗
Android.database.sqlite.SQLiteDatabase	insert∗
Android.database.sqlite.SQLiteDatabase	replace∗
Android.database.sqlite.SQLiteDatabase	update∗
∗OutputStream, ∗Writer	write∗
∗	save∗

Table 3: Data save methods.

Class	Method
Android.content.SharedPreferences	get∗
Android.os.Bundle	get∗
Android.database.sqlite.SQLiteDatabase	query∗
Android.database.sqlite.SQLiteDatabase	rawQuery∗
∗InputStream, ∗Reader	read∗
∗	restore∗

Table 4: Data restore methods.

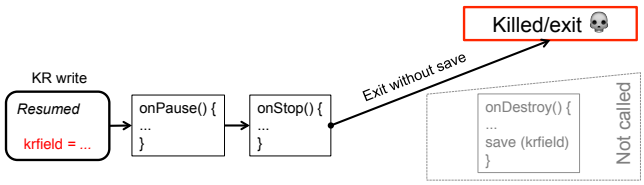


Figure 5: Type 1 loss.

Type 1	May	Forward
Type 2		
First stage	Must	Forward
Second stage	Must	Backward
Type 3	May	Forward
Type 4	May	Forward

Table 5: Dataflow analyses for each error type.

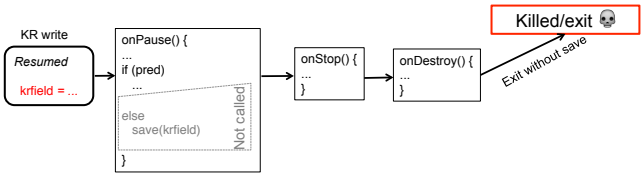


Figure 6: Type 2 loss.

3.3 Defining and Finding KR Errors

KR data saves and restores are critical, as their absence (or presence on paths that are not reached at runtime), can lead to KR errors. While developers generally do write code to handle KR data, the handling can be inconsistent—missing KR saves, KR restores, or both—leading to KR errors. We identify potential errors using data-flow and control-flow analysis.

Notation. We use the following notation: D is the set of KR data fields, i.e., a subset of the mutable fields, as described in Section 3.1. C is the set of field-changing instructions $c_1, c_2, \ldots$, i.e., all those bytecode instructions that mutate a field (or alias thereof). S are field-saving instructions, i.e., those instructions $s_1, s_2, \ldots$ invoking the save methods listed in Table 3; similarly, R are the field-restoring instructions $r_1, r_2, \ldots$ that invoke the restore methods listed in Table 4. G is the set of methods that are guaranteed to be called when the app is winding down, e.g., onPause(); U is the set of methods that are not guaranteed to be called, as per Table 2.

Intricacies of Android/Java static analysis. Since our analysis operates on real-world Android apps, it is natural to see aliasing (our analysis is context-, flow-, field-, object-sensitive thanks to FlowDroid, see Section 4.1). Therefore, since fields d can be manipulated either directly or through aliases, in the reminder of the paper, by "a field d" we mean "d or an alias thereof". Similarly, by "a change c" (or "a save s") we mean changes (or saves) made directly or through an alias. Alias-based changes and saves complicate the analysis as they might introduce additional Gen or Kill dataflow facts on, say, field d, even though the analyzed expression does not contain d.

We divide KR error reports into four loss types. We now define these types and discuss how our approach identifies them; the type and direction for each of these dataflow analyses are shown in Table 5. For Type-1, Type-3, Type-4 and the first stage of Type 2 errors, we essentially use a reaching definitions analysis [4] implemented on top of Flowdroid to compute the possible data flows between restores, changes, and saves.

Type 1. *Definition:* A KR field has only one save operation that is placed in methods that are not guaranteed to be called by the AF, hence the KR field value might be lost. This is shown in Figure 5 where a field krfield is modified while the app is running (i.e., in the *Resumed* state), and its save operation is placed in onDestroy; if the app is killed after onStop(), we have an exit without save, hence krfield is lost. *Identification:* we say KR field d has a potential loss Type-1 if d is changed by at least one instruction c, has at least one save (that is, a KR field d's change c flows to a save s), but *all* saves S are in methods from set U (not guaranteed to be called).

Type 2. *Definition:* A KR field has at least one save operation, but there exists a path from a data change instruction to an activity or app exit AND the path does not contain any save operation for the data; hence the KR field value might be lost. This is shown in Figure 6 where krfield is modified while the app is running and its save operation is placed in onPause(), which is invoked, but due to branch condition pred being true, control flow will follow the then branch, hence again we have an exit without save, and krfield is lost. *Identification:* we say KR field d has a potential loss Type-2 if d is changed by at least one instruction c, has at least one save s where s is in a method guaranteed to be called g following the write to d, but s is not reachable on all paths from c — meaning there exists a path where the change will be lost.

We discover Type-2 errors using a two-stage analysis. In the first stage we use a must-forward analysis to find those changes c to KR fields d that flow into saves s=save(d'); note that the save methods are not necessarily invoked on d – a save could be invoked on field d' (where $d' \neq d$) and the save is still OK as long as d flows (is copied) into d'.

The second stage bears some similarity with *very busy expressions (VBE)* analysis [4], with two differences, as explained next. Recall that an expression e is very busy at point p if for all paths starting at p and ending at the end of the program, an evaluation of e appears before any redefinition of e. So for finding Type-2 errors, one strategy would be to consider changes c (of KR fields d) as the program points p, and saves s as expression uses e; then declare a Type-2 error if c is not very busy (not saved on all paths to the exit). Our anal-

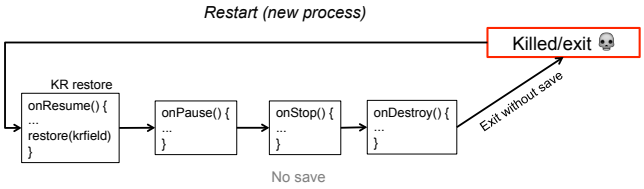


Figure 7: Type 4 loss.

ysis extends the aforementioned strategy in two directions, to deal with the intricacies of Android static analysis: (1) aliasing-based accesses, changes and saves, as discussed earlier in this section; and (2) unlike VBE for which all methods are "equal", our analysis has to discard those saves s placed in methods u that are not guaranteed to be called. This second-stage analysis is must-backward (as VBE is).

Type 3. *Definition:* A KR field has no save and restore operations. If there exists a statement that changes the field, the changed value will be lost, but a KR error does not necessarily occur (e.g., when the post-restart application state is not data- or control-dependent upon this field). *Identification:* there is no dataflow from the change c to either a save s or a restore r.

Type 4. *Definition:* this loss occurs when a KR field does not have a save operation, but has a restore operation, e.g., when the developer forgot to invoke a save method. If the KR field is restored and then changed, we infer that it should have been saved. This is shown in Figure 7 where `krfield` is restored after restart, in the `onRestore()` method, but `krfield` is never saved prior to restart. *Identification:* the restore r is performed in a method guaranteed to be called upon app start, e.g., `onStart()` or `onResume()`, and there exists dataflow from the restore r to a change c.

Termination: We now discuss why the data-flow analyses we have presented, which find the four types of errors, terminate. For Type 1, Type 3, and Type 4, they terminate by virtue of reaching definitions termination [4]. For the first stage of Type 2 errors, the analysis is performed by calling Flowdroid data-flow analysis functions which again terminate [12]. For the second stage of Type 2 errors, our VBE variant terminates because the dataflow analysis uses monotonic transfer functions on a finite lattice. The lattice has just two elements: true and false (i.e., whether the field has changed or not). The initial state is true. Then, all of the subsequent flow functions either leave it as true or modify it to false, hence the flow functions are monotonic. As the height of the lattice is finite (i.e., 1), the algorithm therefore terminates [26].

3.4 Reporting and Reproducing Potential Bugs

After finding the potential losses, we produce *error summaries*, i.e.,

1. Error type, KR field name, location (i.e., the method and change instruction for that KR field).
2. A summary of the control flow path that leads to the error.

Given these summaries, we can move to reproducing and confirming the errors. For this, we use a suite of static analyses (provided by other tools) to construct *directed transitions*, i.e., sequences of GUI events and method callbacks that must be invoked to drive the execution to the error location. Once the app is in the error location, we can go through pause-and-resume (or kill-and-restart) cycles to confirm the error. This suite relies mostly on third-party tools, so we describe it in the implementation part (Section 4.2).

4. Implementation

We now briefly describe our static analysis implementation, as well as the implementations associated with error reproduction and error report verification.

4.1 Static Analysis with *KREfinder*

KREfinder is based on Flowdroid [12] (an open-source static analysis tool for Android apps) version 1.0. The analysis is context-, flow-, and object-sensitive; the points-to analysis is Andersen [5] (Flowdroid defaults).

Limitations. *KREfinder* inherits Flowdroid's soundness goals: sound up to reflective/native calls. Also, while Flowdroid conservatively handles Android-specific, heavily-used constructs, such as Intents and Broadcasts via overapproximation, it does not handle implicit flows, hence *KREfinder* does not handle implicit flows either.

We now provide a brief description of Flowdroid and our extensions to it that constitute *KREfinder*. Flowdroid's main goal is static taint analysis, i.e., determining whether a tainted "source" reaches a security-sensitive "sink". We have extended Flowdroid in two main directions — fine-grained flow tracking and scalability — to fit our requirements, and then added several new implementation modules for finding KR errors.

First, Flowdroid only supports *marking methods* as sinks and sources, *not arbitrary instructions*. This hinders the process of finding KR errors, since finding such errors requires the ability to trace dataflow between instructions (including call statements, assignment statements, exit statements, and return statements). We thus extended Flowdroid to be able to trace flow between arbitrary instructions.

Second, Flowdroid requires large amounts of memory for large apps—while scalability can be improved by setting certain command-line arguments, increasing scalability will reduce precision. We discovered that the large memory footprint is due to Flowdroid simultaneously tracking multiple pairs of sinks and sources. Hence we came up with an intuitive, yet crucial modification: instead of tracking multiple sink-source pairs simultaneously, we changed Flowdroid to trace flow from one source only at a time, and run Flowdroid multiple times with different sources. This change of flow tracking strategy has improved scalability significantly.

```
====== path ======
      Lorg/hermit/audalyzer/Audalyzer;→
                        onCreate(Landroid/os/Bundle;)V
-##→ Lorg/hermit/audalyzer/Audalyzer;→
       onOptionsItemSelected(Landroid/view/MenuItem;)Z
-->  Lorg/hermit/audalyzer/Audalyzer;→ showAbout()V
-->  Lorg/hermit/android/core/MainActivity;→ showAbout()V
-->  Lorg/hermit/android/core/MainActivity;→ createMessageBox()V
-->  Lorg/hermit/android/core/AppUtils;→ getVersionString()
                                    Ljava/lang/String;
-->  Lorg/hermit/audalyzer/Audalyzer;→ showFirstEula()V
-->  Lorg/hermit/android/core/MainActivity;→ showFirstEula()V
-->  Lorg/hermit/android/core/OneTimeDialog;→ showFirst()V
-->  Lorg/hermit/android/core/OneTimeDialog;→ isAccepted()Z
```

Figure 8: Directed path transitioning to the method getVersionString(), as produced by Redexer.

```
      <View type="android.view.MenuItem" id="2131099661" idName="menu_eula"
title="Disclaimer">
        <EventAndHandler event="item_selected"
handler="&lt;org.hermit.audalyzer.Audalyzer: boolean
onOptionsItemSelected(android.view.MenuItem)&gt;" />
      </View>
```

Figure 9: Our toolset, based on output from Redexer and Gator, reveals the Disclaimer menu item as the GUI element to invoke to reach this view.

After these extensions, we implemented the Section 3 algorithms as several new modules on top of Flowdroid, yielding *KREfinder*, a static analysis that reveals potential KR errors. Each error report includes the change statement and its location (method name, class name, package name, save statement and restore statement), as well as summaries of data- and control-flow (from change to save or exit).

4.2 Verifying and Reproducing KR Errors

After *KREfinder* completes, producing a set of potential bug reports, our *KREreproducer* helps reproduce and verify the errors. *KREreproducer* uses a combination of new and extended tools, consisting of mainly two steps: First, creating a directed transition to the error point on legal app execution paths. Second, forcing the app to exit, to mimic an actual scenario, either pause-and-resume or kill-and-restart, and observe the results.

4.2.1 Example: Reproducing Audalyzer's KR Error

We start with an example, the Audalyzer app, to demonstrate how we verify the potential bug reports produced by the static analysis. Audalyzer is an audio analyzer. When the app is first opened, an End User License Agreement (EULA) is displayed; the user accepts it by pressing the 'Accept' button, and the app sets the isAccepted field to **true**. This ensures that the EULA confirmation will never pop up again. But if the app exits just after the user presses 'Accept', the isAccepted field value change might be lost, as it is not saved on all exit paths.

KREfinder reported this error, indicating that isAccepted was changed in the isAccepted() method of the org/hermit/android/core/OneTimeDialog class. Hence, to reproduce the error we need to trigger input events in such a way that we execute this method and then exit the app at the point after the change.

To do so, *KREreproducer* uses the Redexer binary rewriting infrastructure [25] to generate a *directed path transition*. Given the Audalyzer.apk and the target method isAccepted(), Redexer produces a sequence of callbacks and associated method calls such that calling the sequence will lead to isAccepted() being invoked. Figure 8 shows the corresponding output of Redexer for this scenario: the types of callbacks that need to be generated to end up in the isAccepted() method.

Note, however, that just generating the sequence of callbacks is not enough. We need to identify the associated GUI elements that, when exercised by the user, trigger those callbacks. To do so, *KREreproducer* uses another tool, Gator [33], which, given an APK file, produces the necessary GUI element-callback mapping used in the program. Coupling Gator's and Redexer's outputs we can identify which GUI elements we need to exercise to reach the point of interest in the execution path for invoking the desired method. Figure 9 shows one of the outputs of Gator after we filter out results. From Figure 8 we see that the first GUI callback event was onOptionsItemSelected in the org/hermit/audalyzer/Audalyzer class. Figure 9 shows the output of our toolset, revealing that the associated GUI element is a menu item with the title 'Disclaimer'. That is, we need to first invoke the application menu and select the 'Disclaimer' item. We then automate GUI interactions based on these results, using the A^3E [13] app exploration tool: given the sequence of GUI element names, A^3E exercises the sequence and "lands" the app at the error point, with no user intervention.

After reaching the error point at runtime, we killed Audalyzer, i.e., forced the app onto an exit path. Finally, we restarted the app to verify whether the changes made in 'isAccepted' were persistent, i.e., the value of the isAccepted field was still **true** (it was not—this is precisely the problem). We now describe the general techniques for achieving directed transition and exiting.

4.2.2 Directed Transition to the Error Point

The directed transitions generated by *KREreproducer* will guide app execution to error points. While generating inputs to direct execution to a certain program point is of course undecidable in the general case, the GUI-oriented, event-based nature and shallow stack of Android apps make it easier in practice to reach a certain point in a certain method.

Generating the directed transition path. Based on *KREfinder*'s output, we identify the methods we are interested in. Next, *KREreproducer* uses Redexer to generate a sequence of events, starting from the app's home screen that will provide a directed transition to the method specified.

Creating sets of valid "View-Events" pair. Given the aforementioned sequence of events, *KREreproducer* maps the events with required View elements. To do this, *KREreproducer* uses Gator which conducts static reference analysis to create a one-to-one mapping from View elements to their associated callbacks; we now have a sequence of callbacks ready to be fired.

(Optional) Automated exploration. With the event and callback sequences at hand, the developer can use the A^{3}E explorer to automatically guide the app into the desired state; alternatively, the user may attempt to fire the events manually by exercising the corresponding UI elements.

4.2.3 App Resume/Restart and Monitoring

Once the app is in a potential-error state (it has made changes to KR data but the changes are not saved) we need to drive the app down an exit path, which varies depending on the desired restart level. We triggered restarts at the three levels defined in Section 2.2/Table 1, i.e.:

1. Pausing the app by popping up a dialog-themed activity, which invokes onPause().
2. Stopping the app by pressing the 'Home' button, which invokes onStop().
3. Kill (finish) the app by pressing the 'Back' button, which invokes onDestroy().

After the app exits, we restart it. Upon restart, one of the error categories described in Section 5.2.1 will materialize, or the app might even crash outright.

To get a hold of the error details, we track individual app fields (KR data) that the static analysis has identified as potentially being lost, as follows: we insert calls to log field values before and after the exits, using the apktool [2]. To dump GUI states (so we can compare the before-restart and after-restart GUI states and find differences) we use Redexer to rewrite the app to dump GUI states before exits and after resume/restart. Finally, we monitor app execution for crashes via the logcat Android system log utility.

To conclude, we have constructed an end-to-end system that, given an Android app as input, will automate KR error discovery, location, and reproduction.

5. Evaluation

We have evaluated our approach along two main dimensions. First, effectiveness: *can the approach run on popular, off-the-shelf apps?* and *is the approach effective at finding potential bugs?* Second, efficiency: *does the analysis complete in a reasonable amount of time?* and *is the process burdensome for users?*

5.1 Methodology

App datasets. We downloaded and examined 334 apps from Google Play (`play.google.com`), AppsApk (`www.appsapk.com`), and Google Code (`code.google.com`). We chose these apps using several criteria which we believe are necessary for making meaningful observations. In particular, the 334 applications: (a) cover different categories, e.g., Utilities, Email & SMS, Games, Health & Fitness, Wallpapers, Photography, Weather, News, Education, Browser, Map, Call & Contacts; (b) have variety in terms of size, from 3KB to 26MB; and (c) have variety in terms of popularity, e.g., many apps have over 1 million installs while Facebook has over 1 billion installs.

Criteria	# apps	Tag
Initial set	334	
Can be analyzed with Flowdroid	324	SA-324
Have potential errors	114	
Decompilable	109	
Decompilable&have potential errors	49	
Have true positives	37	TP-37

Table 6: Datasets construction.

From these 334 apps, we constructed two datasets of 324 apps and 37 apps, respectively (SA-324 and TP-37), as shown in Table 6 and explained below. Of the 334 apps, 324 could be successfully analyzed with Flowdroid; we name this dataset SA-324. Among these 324 apps, our static analysis reported potential errors in 114. Next, we attempted to decompile the 324 apps using the Dare [30] and Dex2jar [15] decompilers; from 324 apps, only 109 could be decompiled successfully.[6] Out of the 109 apps that could be decompiled, the static analysis reported potential errors in 49. With the decompiled source code for the 49 apps at hand, we manually analyzed the source code to discern false positives from true positives. We found that 37 apps had true positives—these constitute the TP-37 dataset. Our evaluation henceforth proceeds on the SA-324 and TP-37 datasets.

5.2 Effectiveness

SA-324. The app characteristics and overall results for the 324 apps are presented in Table 7. The largest app in the dataset, Facebook, was about 23 MB in size, while the median app size was 168 KB. The analysis found potential errors in 114 out of 324 apps, hence the average 0.8 reports for the entire dataset. Across those 114 apps with potential errors, there were 2.3 error reports per app, on average.

TP-37. Table 8 shows app characteristics and analysis results for the apps in TP-37. For each app we provide the name, version, size, popularity (number of installs per Google Play or downloads per Google Code), static analysis time, the number of true positives and the number of false positives.

The true positives were confirmed using the error reproduction process described in Section 4.2. While the number

[6] Decompilation can fail due to several reasons, e.g., apps use obfuscation, or the decompiler is brittle.

Size (KB)				Time (sec.)				Reports			
min	max	average	median	min	max	average	median	min	max	average	median
3	23,112	377	168	1	2,184	61	22	0	6	0.8	0

Table 7: App characteristics and analysis results for dataset SA-324.

App Name	Version	Size (KB)	Installs	Time (sec.)	True Positives	False Positives
1. Dr.WebAnti-virusLight	v.9	1,464	50,000,000–100,000,000	60	1	3
2. Motorola Camera	3.1.5	2,529	10,000,000–50,000,000	948	1	2
3. Alarm Clock Plus	4.7.1	2,200	5,000,000–10,000,000	76	1	2
4. FoxFi	2.17	388	5,000,000–10,000,000	31	2	1
5. Souvey Musical Pro	6.0.7	776	5,000,000–10,000,000	75	3	1
6. OI File Manager	2.0.2	927	5,000,000–10,000,000	54	1	1
7. OpenSudoku	1.1.5	219	1,000,000–5,000,000	102	1	1
8. Open WordSearch	2.3.4	680	1,000,000–5,000,000	75	1	0
9. NPR News	2.1	543	1,000,000–5,000,000	36	1	0
10. Painless Power Toggles	5.0.03	624	1,000,000–5,000,000	37	3	3
11. Symantec Norton Snap	1.0.1.62	1,606	1,000,000–5,000,000	30	1	1
12. Alarm Klock	1.9	340	500,000–1,000,000	44	1	1
13. MOBILedit! PC Suite	1.1.19	133	500,000–1,000,000	19	2	1
14. ScrambledNet	3.4	259	500,000–1,000,000	1,569	1	1
15. Phone Copier	3.0.2	662	100,000–500,000	156	1	0
16. Dock4Droid	3.5	268	100,000-500,000	84	1	1
17. MiniFetion	2.8	129	100,000–500,000	121	1	0
18. Scrollable News	2.0.0	760	100,000–500,000	165	1	1
19. Zirco Browser	0.3.2	210	50,000–100,000	71	1	3
20. Brightness Profiles	1.3	25	50,000–100,000	20	1	1
21. Speaker Proxmty donate	0.3.5b	117	12,300	25	1	1
22. AndDaaven Siddur	0.2.6	645	10,000–50,000	8	1	0
23. Droidstack	1.0.11a	224	10,000–50,000	47	1	2
24. Open Live View	1.0.2.2	440	10,000-50,000	79	1	1
25. SSH Tunnel	2.0.3	618	10,000–50,000	77	3	0
26. BTHF PowerSave	1.0	77	10,000–50,000	23	3	0
27. DiskDigger Pro	1.0	978	10,000–50,000	64	1	1
28. ServDroid.Web	0.1	156	10,000–50,000	26	1	0
29. DateSlider	1.2	54	10,000	37	1	2
30. Copter GCS	8	284	8,300	24	3	2
31. CalenMob	1.0.3	484	5,000-10,000	105	1	1
32. AndroidToken	2_01	104	4,000	111	1	4
33. BlueNET	0.1	108	4,200	17	1	2
34. VPN_Connections	v04	395	3,000	19	1	1
35. BikeRoute	1.0.3.7	446	3,000	137	1	2
36. KITCard Reader	0.9	33	1,000–5,000	11	1	0
37. Personal Work Recorder	0.0.4	73	500	28	1	2

Table 8: App characteristics and analysis results for dataset TP-37.

of false positives can be further reduced by improving the precision of the static analysis (Section 5.2.2), we found the process of triaging to be quite efficient, 5–15 minutes per error report and reproduced bug, as explained in detail in Section 5.3; hence even for apps with a high positive rate, e.g.,

App	Error description	KR field(s) lost	Restart kind
1. Alarm Clock Plus	The user sets the alarm; after restart the alarm is not set any longer	mSelectedId	3
2. Alarm Klock	The user changes the time of an alarm; after restart the alarm time change is gone	time	2
3. AndDaaven	The user navigates through a set of prayer lines. The last position is lost after restart	currentOffset	3
4. Android Token	Users selects an existing token for a one-time password. After restart the selection is lost	mSelectedTokenId	3
5. BikeRoute	A list of routes was populated. Upon restart, the current selected route is lost	id	3
6. BlueNET	The user turns the gateway server on. The server is turned off after restart	state	3
7. Brightness Profiles	Brightness level is lost after restart	appBrightness	3
8. BTHF PowerSave	The user changes the states of the master switch service, bluetooth module and outgoing calls. After restart, the changes are lost	switchOffBTAfterCall Ended, enabled, processOutgoingCalls	3
9. CalenMob	The user sets the working date to a future date. After restart, the working date resets to current date	AgendaStart	3
10.Copter GCS	This app is a Ground Control Station for the arduCopter and arduPirate Unmanned Aerial Vehicles. The user changes a set of states in Mode Selection and Readouts, e.g., Gyro X, Accel Y, then saves them. After restart, the state changes are lost	spinners, sb, temp	3
11.DateSlider	The chosen date is lost after restart	mTime	3
12.DiskDigger Pro	The user sets the percentage to start scanning from; percentage is lost after restart	"long c"	3
13.Dock4Droid	This is a home screen dock app. Users can stick their favorite apps as launcher icons in the dock and open them directly from the dock. To do so users have to go to the 'settings' menu add apps to the dock. After restart the last-added app is missing from the dock	name	3
14.Dr.WebAnti-virusLight	The custom scan check box setting is lost after restart	java.lang.String [] g	3
15.Droidstack	This is a Stack Exchange client app. Upon selecting a stack exchange post, the title is not preserved after restart	title	3
16.FoxFi	The user enters email and serial number. After restart the entered data is lost	aM, aS	2
17.KITCard Reader	This is a magnetic card reader app. After restart the last-fetched balance is lost	lastBalance	3
18.MiniFetion	User composes an SMS message and pauses; after resuming the message is lost.	mMessage	1
19.MOBILedit! PC Suite	After install, if the application is allowed to connect to a local WiFi, a restart does not retain the WiFi settings	protocolVersion, mAllowRemote	3
20.Motorola Camera	The user switches from image mode to video mode, then stops and resumes. The camera returns to image mode again	mMode	2
21.NPR News	While navigating through stories, if a user selects a story, and the app restarts, the selected story was not loaded after a restart	timestampFirst	3
22.OI File Manager	After selecting a file and restarting, the last-opened path is lost	mPath	3
23.Open Live View	This is a smart watch app. Recent changes to the connected device list are lost after restart	valueList	3
24.OpenSudoku	Game state (user-filled numbers) lost after restart	mValue	3
25.Open WordSearch	The user fills a new word and then restarts. The new word is gone.	storeId	2
26.Personal Work Recorder	If restarting while the app is recording a workout session, the workout start time is lost	start_at	3
27.Painless Power Toggles	The appearance of notification widgets is lost after restart	icon, flags, bigContentView	3
28.Phone Copier	After restart, the email address entered by the user is lost	mEmail	3
29.Scrambled Net	The game state is lost after a restart	gameState	3
30.Scrollable News	The user selects a color. The selection is lost after a restart	selectedPosition	2
31.ServDroid.web	The user changes the status and then restarts. The new status is gone.	mSettings	2
32.Souvey Musical Pro	This is a musical toolkit including instruments (piano, drums, autoharp), tools (metronome, pitch pipe) and a game. The user changes the settings of Metronome and then stops the app. After resuming, the changes are lost.	bpm, beats, notes	2
33.SpeakerProximity donate	The app automatically turns the speaker phone on and off based on input from the proximity sensor. After restart the sound settings are lost	value2	3
34.SSH Tunnel	SSH connection profile (local and remote port numbers, server addresses) is lost upon restart	localPort, remotePort, hostIp	3
35.Symantec Norton Snap	User turns on the flashlight, then stops the app; after resuming, flashlight is off	"boolean d"	2
36.VPN Connections	The user changes IPSec ID, but the ID lost upon restart	ipSecId	3
37.Zirco Browser	Bookmark lost after restart	mDbHelper	3

Table 9: Description of reproduced bugs for dataset TP-37.

AndroidToken, we believe that the error reproduction burden is acceptable.

In Table 9 we provide a detailed description of the 49 bugs we reproduced: app name, a brief description of the error, the name of the lost KR field(s) and the restart kind (1, 2, or 3, defined at the beginning of Section 4.2.3). As we can see, the error manifestations vary depending on the app. E.g., the alarm time gets lost after it is set in Alarm Clock Plus; the Personal Work Recorder app fails to keep track of the workout start time; the Zirco browser loses the bookmark the user has just set.

Many of the errors we present in Table 9 can be found in the apps' reviews on Google Play and Google Code, e.g.:

Brightness Profiles: "When application window disappears brightness returns to previous level"; "As soon as you exit to home screen brightness level jumps back up to phone's minimum level"; "the lowest brightness level doesn't stay dim, even w/auto-brite off"; "Randomly sets back to 100% and auto brightness after using camera" [18].

SSH Tunnel: "Enable ssh tunnel. connects fine. when wireless connection is lost and reconnected ssh tunnel does not reconnect"; "ssh tunnel is turned disabled/unchecked. Auto connect and reconnect are still checked" [17].

Souvey Musical Pro: "This app is great, but there is one thing that has been frustrating me with it: if the metronome is stopped and the phone locks, when I unlock it again the metronome has gone back to 40 bpm. Any chance of a fix?"; "On Droid 2, the metronome is frustrating. I use it when I practice but if I ever stop it, and turn off the screen, it resets to 40bpm and 4/4 time." [19].

Scrambled Net: "1. Play Game, get near end, just about ready to solve difficult problem; 2. Boss calls; 3. Go back to game later; 4. Game not (always) 'paused', need to start new (swear)." [16]

5.2.1 KR Error Manifestation Categories

Based on the reproduced errors, we now categorize KR errors according to their manifestations.

Losing GUI state. While Android supports saving GUI state across restarts via the default implementation of onSaveInstanceState (Section 2.4), GUI state loss can still occur for two main reasons: onSaveInstanceState() is not invoked, or the app has a custom GUI that is not managed by the AF. Examples of unsaved GUI state include the app "forgetting" the custom scan setting in Dr.WebAnti-virusLight.

Losing user's progress. In many cases, KR errors lead to losing user's work (e.g., the message draft is lost when the user composes a message in MiniFetion, the Zirco Browser app "forgets" to save a bookmark before restart), play (e.g., game state is lost in OpenSudoku and Scrambled-Net), or workout data (the workout timing in Personal Work Recorder).

Losing device settings. Many KR errors lead to device setting being lost: the phone's flashlight setting is lost in Symantec Norton Snap; the speaker phone setting is lost in SpeakerProximity; the camera setting is lost in Motorola Camera; the Bluetooth settings are lost in BTHF PowerSave and FoxFi.

App Name	Size (KB)	Installs	Time (sec.)	Reports
Facebook	23,112	1,000,000,000–5,000,000,000	35	0
UC Browser	13,429	100,000,000–500,000,000	21	0
Media Mogul	5,617	1,587	12	0
Alarm Clock Plus	2,245	5,000,000–10,000,000	76	3
Symantec Norton Snap	1,606	1,000,000–5,000,000	30	2

Table 10: Top-5 largest apps.

5.2.2 False Positives

As with any static analysis, our approach is prone to scalability issues and false positives. We have identified four main sources of false positives.

First, many false positives are potential loss type 3 (Section 3.3). We include them to preserve our soundness guarantees, but in practice we found them to be benign. That is, a type 3 potential loss does not lead to post-restart errors due to lack of dependencies from the unsaved fields to post-restart code. A further control and data dependency analysis could be used to reduce the false positive rate; we leave this to future work.

The second source is flow imprecision. Flowdroid's "dummy main" function that models how callbacks will be invoked has limited precision: it assumes callbacks can be invoked in any order (hence potentially introducing spurious data flow), but in practice the order is constrained. This imprecision could be alleviated by incorporating ideas from other static analyzers such as Gator [33].

Third, an app may offer GUI options for saving data, e.g., a 'Save' button. Our static analysis does not identify these, and assumes the data will not be saved; e.g., the app Alarm Clock Xtreme had one such false positive.

Fourth, imprecision in Soot's alias analysis (the alias analysis used by Flowdroid) leads to false positives.

5.3 Efficiency

We ran our static analysis on an 8-core Intel Xeon i7-4770 (8MB Cache, 3.4 GHz) with 32GB of RAM. The system ran Ubuntu 14.04.1, Linux kernel version 3.13.0-32-generic.

Static analysis efficiency. SA-324. Running the static analysis on the entire dataset took 6 hours. The "Time" column in Table 7 shows running time statistics: the mean

App Name	Size (KB)	Installs	Time (sec.)	Reports
Painless Power Toggles	624	1,000,000–5,000,000	36	6
Copter GCS	284	8,300	24	5
AndroidToken	104	4,000	111	5
Dr.WebAnti-virusLight	1,464	50,000,000–100,000,000	60	4
TheElements	441	1,500	40	4

Table 11: Top-5 apps in terms of error reports.

App Name	Size (KB)	Installs	Time (sec.)	Reports
ScrambledNet 3.3	247	500,000–1,000,000	1,569	3
Motorola Camera	2,529	10,000,000–50,000,000	948	3
Yuchdroid	328	2,350	670	1
EmWeather	607	1,000	456	0
Audalyzer	196	1,000	256	4

Table 12: Top-5 apps with highest analysis time.

App Name	Size (KB)	Installs	Time (sec.)	Reports
Facebook	23,112	1,000,000,000–5,000,000,000	35	0
UC Browser	13,429	100,000,000–500,000,000	21	0
Dr.WebAnti-virusLight	1,464	50,000,000–100,000,000	60	4
Yahoo weather	391	10,000,000–50,000,000	10	0
Alarm Clock Plus	2,245	5,000,000–10,000,000	76	3

Table 13: Top-5 most downloaded apps.

analysis time is 61 seconds while the median is 22 seconds, which shows that our analysis is practical.

TP−37. Table 8's "Time" column shows the individual running time for the 37 apps with true positives. While times ranges from 8 to 1,569 seconds, the analysis typically completes in less than 5 minutes.

Input generation efficiency. The input generation stage took less than a minute for the examined apps; we omit individual numbers for brevity.

Error report triaging (manual effort) efficiency. While we did not precisely measure the time it took to triage the error reports in the manually-analyzed apps, the first author, as well as the second author, were consistently able to triage each error report as follows. Ruling out a report as a false positive takes less than 5 minutes. Understanding and reproducing a bug takes about 10-15 minutes; of course, in case of multiple error reports, the cost for triaging subsequent reports is lower due to the initial learning curve. Note that the apps were unfamiliar to the triagers, so we expect the time to further reduce when developers run our analysis on apps they are familiar with.

Top-5 apps. We now present the results of running the analysis for top-5 apps according to several criteria. The characteristics and results for the largest apps are presented in Table 10: while analysis time depends on many factors (aliasing, size of control flow graph, etc.) note how our approach can handle substantial APKs. The top-5 apps in terms of errors are shown in Table 11; this shows the precision of our analysis, as we believe that triaging 4–6 reports is quite manageable. The apps with the highest analysis time are shown in Table 12; we believe that even the longest times, e.g., 1,569 or 948 seconds for ScrambledNet 3.3 and Motorola Camera respectively, are acceptable. The most popular apps, all exceeding 5 million installs, are shown in Table 13; this illustrates the applicability of our analysis on popular apps.

Summary. We are now in a position to answer the questions set forth at the beginning of this section. Our approach is effective: it handles sizable off-the-shelf apps without requiring access to the source code; has uncovered dozens of errors in popular apps; and has a low false-positive rate. Our approach is efficient: the static analysis part typically completes in 61 seconds; input generation takes less than a minute as well, which allows users to examine reports as well as reproduce and locate errors efficiently. The manual analysis process typically takes 5–15 minutes per error report, which suggests that the analysis is effective at finding, triaging and reproducing errors.

6. Related Work

Resume and restart errors. Zaeem et al. [37] have developed an app testing tool named QUANTUM. QUANTUM uses a library of "interaction features" (common patterns of users interacting with apps), to test apps by constructing parsimonious test cases from GUI models and interaction features; QUANTUM compares GUI state before and after features are exercised. Their features include screen rotation (which involves a stop/restart per Section 2.3), killing and restarting the app, pausing and resuming the app, and pressing the 'Back' button. They tested 6 apps and found 22

bugs, including kill&restart as well as pause&resume bugs. QUANTUM and our approach differ most fundamentally in the traditional testing vs. static analysis way. Specifically: we use a sound static analysis whereas QUANTUM is dynamic; by comparing GUI states they find only those manifestations of KR errors that are reflected in the GUI, whereas our approach can find those and more subtle losses that might not be visible in the GUI; our approach is fully automatic whereas QUANTUM requires a GUI model to be manually constructed and manually validated by the user; we focus exclusively on resume-and-restart errors, whereas their focus is on generating parsimonious testing suites from a library of user interaction features; we focus on batch-analyzing dozens or hundreds of apps without requiring access to app source code. Our Directed Transitions generator produces event sequences, rather than test cases as QUANTUM does, with the goal of hitting a certain program point (rather than achieving coverage as does QUANTUM).

Adamsen et al. [3] use a somewhat similar approach to QUANTUM: given a test suite, they inject "neutral" event sequences including pause-resume, pause-stop-restart, pause-stop-destroy-create (in addition to other audio or telephony neutral sequences) and see if the supposedly neutral sequence lead to test failures. Their approach has found 22 critical bugs (e.g., crash or user setting lost) in 4 apps. Our approach differs from Adamsen et al.'s in a similar way as it differs from QUANTUM: Adamsen et al.'s approach is dynamic and relies on availability of test suites, looking for test suite failures in general, whereas we have constructed: (1) a static analysis meant to batch-analyze apps specifically for resume-and-restart errors without access to test suites and (2) a Directed Transitions generator meant to hit a certain program point and help debug a single error, rather than discover multiple unknown errors as Adamsen et al.'s approach does.

Finding smartphone-specific bugs. AppDoctor [23] tries to find bugs by injecting various events into an app's execution; one such event is "Rotate". They mention the problem of apps having to handle restarts at any moment, but their approach does not focus on this problem, as we do, and instead their goal is to find bugs by rapid firing of events. They found two bugs (in OpenSudoku and Google Translate) where rapid event firing catches the app in an inconsistent state, but the bugs are due to rapid event interleaving rather than state loss upon restart.

Most of the smartphone-specific analysis and verification work so far has focused on other kinds of bugs. Hsiao et al. [21] as well as Maiya et al. [28] have used dynamic analysis to find races in Android apps. Pathak et al. have used dynamic analysis to find energy bugs [31]. Hu et al. have used dynamic analysis to find when apps attempt to deviate from the Activity state machine [22]. Caiipa [27] uses *contextual fuzzing* (e.g., fuzzing network parameters or GPS locations) to discover bugs in mobile apps.

Our work differs in two key regards. First, we focus on a new class of bugs—KR errors. Second, we use static analysis which is sound but incomplete (prone to false positives) whereas dynamic analysis is unsound (prone to false negatives) but complete.

Finding persistency bugs in traditional applications. Prior efforts on persistent data consistency have looked at finding file system [36] or database errors [14]. Yang et al. employ model checking to find errors in filesystem code [36]. Gunawi et al. [20] use checkers based on declarative specifications to check the integrity of, and repair files systems. Subramanian et al. [35] studied the impact of disk data corruption on database integrity.

These techniques though, are not directly applicable in the context of KR data handling on Android due to the significant difference in domains, applications, and consistency properties checked by theirs and our approaches.

Directed testing. While automated testing approaches for Android abound (Ravindranath et al. [32] provide a comprehensive survey), most of them do not aim to "hit" a particular state or program point. Collider [24] uses symbolic execution to reach a target instruction, hence it can generate a precise path condition for reaching the target. While in Collider UI models were built manually, they could be constructed automatically using third-party tools. Collider requires access to the application source code; it is unclear how well the symbolic execution scales to very large apps. Our directed transition method is less precise—it could potentially fail for complicated path conditions, though we have not seen such failures on our dataset. In general, we aim at a different design point: a scalable approach that can be applied efficiently to large popular apps from Google Play via a fully automatic tool-chain and without requiring the source code.

7. Conclusions and Future Work

We reveal a new class of errors, "KR errors," germane to the smartphone platform: loss of state when an app is paused, stopped, or killed. We construct a novel static analysis and tool, *KREfinder*, that combines data- and control-flow analyses to automatically find potential KR errors in Android apps. The error reports are passed to another tool we constructed, *KREreproducer*, which generates input sequences to help users verify the report. Experiments with analyzing 324 Android apps and reproducing bugs in 37 apps have revealed that our approach is effective at finding as well as reproducing errors, and runs efficiently on sizable apps.

We believe that our technique—finding which data survives a restart—opens a more general research direction, exploring the trade-off between over-saving and under-saving; with our technique, app developers/users know exactly what data survives a restart, and whether a restart will lose state or can clean up the state.

Acknowledgments

We thank Gogul Balakrishnan and the anonymous referees for comments on this work. This work was supported in part by a Google Research Award. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] Uiapplicationdelegate protocol reference, March 2016. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/index.html#//apple_ref/doc/uid/TP40006786.

[2] android-apktool: A tool for reverse engineering android apk files, March 2016. https://code.google.com/p/android-apktool/.

[3] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771786. URL http://doi.acm.org/10.1145/2771783.2771786.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[6] Android Open Source Project. Activities, March 2016. https://developer.android.com/guide/components/activities.html.

[7] Android Open Source Project. Managing the activity lifecycle, July 2016. http://developer.android.com/training/basics/activity-lifecycle/.

[8] Android Open Source Project. Platform versions, July 2016. https://developer.android.com/about/dashboards/index.html.

[9] Android Open Source Project. Pausing and resuming an activity, March 2016. https://developer.android.com/training/basics/activity-lifecycle/pausing.html.

[10] Android Open Source Project. Processes and application life cycle, March 2016. https://developer.android.com/guide/topics/processes/process-lifecycle.html.

[11] Android Open Source Project. Saving data, March 2016. https://developer.android.com/training/basics/data-storage/index.html.

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594299. URL http://doi.acm.org/10.1145/2594291.2594299.

[13] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509549. URL http://doi.acm.org/10.1145/2509136.2509549.

[14] P. Bohannon, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. Detection and recovery techniques for database corruption. *Knowledge and Data Engineering, IEEE Transactions on*, 15(5):1120–1136, Sept 2003.

[15] Google Code. Dex2jar. https://code.google.com/p/dex2jar/.

[16] Google Code. Scramblednet issue 54: Scramblednet loses game state, March 2016. https://code.google.com/p/moonblink/issues/detail?id=54.

[17] Google Code. Ssh tunnel issue 165: Auto connect and auto reconnect not working, March 2016. https://code.google.com/p/sshtunnel/issues/detail?id=165.

[18] Google Play reviewers. Brightness profiles, March 2016. https://play.google.com/store/apps/details?id=com.angrydoughnuts.android.brightprof.

[19] Google Play reviewers. Souvey musical pro, March 2016. https://play.google.com/store/apps/details?id=com.angrydoughnuts.android.brightprof.

[20] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Sqck: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 131–146, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855751.

[21] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594330. URL http://doi.acm.org/10.1145/2594291.2594330.

[22] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-

0592-1. doi: 10.1145/1982595.1982612. URL http://doi.acm.org/10.1145/1982595.1982612.

[23] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 18:1–18:15, 2014.

[24] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, 2013.

[25] Jinseong Jeon and Kristopher Micinski and Jeffrey S. Foster. Redexer. http://www.cs.umd.edu/projects/PL/redexer/index.html.

[26] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977. ISSN 1432-0525. doi: 10.1007/BF00290339. URL http://dx.doi.org/10.1007/BF00290339.

[27] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 519–530, 2014.

[28] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594311. URL http://doi.acm.org/10.1145/2594291.2594311.

[29] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, Sept. 2000. ISSN 0360-0300. doi: 10.1145/367701.367728. URL http://doi.acm.org/10.1145/367701.367728.

[30] D. Octeau, S. Jha, and P. McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1–6:11, 2012. ISBN 978-1-4503-1614-9.

[31] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1301-8. doi: 10.1145/2307636.2307661. URL http://doi.acm.org/10.1145/2307636.2307661.

[32] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 190–203, 2014.

[33] A. Rountev and D. Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, 2014.

[34] Stack Overflow. Android activity life cycle - what are all these methods for?, March 2016. http://stackoverflow.com/questions/8515936/android-activity-life-cycle-what-are-all-these\-methods-for.

[35] S. Subramanian, Y. Zhang, R. Vaidyanathan, H. Gunawi, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and J. Naughton. Impact of disk corruption on open-source dbms. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 509–520, March 2010. doi: 10.1109/ICDE.2010.5447821.

[36] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, Nov. 2006. ISSN 0734-2071.

[37] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 183–192, March 2014. doi: 10.1109/ICST.2014.31.