Detection of Energy-Inefficiency Patterns in Android Applications

Dissertation

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Haowei Wu
Graduate Program in Computer Science and Engineering

The Ohio State University

2018

Dissertation Committee:

Atanas Rountev, Advisor

Michael Bond

Neelam Soundarajan

ABSTRACT

Android has become the most widely used mobile platform. Android developers continuously introduce new features to their applications, which are consequently becoming increasingly complex. This complexity could lead to a variety of software defects. Due to the limited battery capacity of Android mobile devices, *energy-related defects* are of significant importance to developers and app users. This motivates our work to focus on detecting energy-inefficiency patterns on Android apps.

One important source of energy inefficiency is *missing deactivation* of energy-related resources. Such resources are acquired during execution and continue to be held by the app even after they are not being used. As the first contribution of this dissertation, we present a static analysis to detect energy leaks caused by GPS listeners. Such listeners keep the GPS active and can cause considerable battery drain. Existing work on similar problems is mainly based on static or dynamic program analysis. For dynamic analysis, one shortcoming is that it is difficult to achieve high code coverage, and code regions that contain energy-related leaks might not be executed. For static analysis, one difficulty is that Android apps are event-driven and their run-time behavior depends on the sequence of callbacks from the Android framework to app. Such callback sequences are not modeled effectively by existing static analyses. We aim to solve this problem by using a comprehensive static model of Android GUI behavior, and paying careful attention to feasibility constraints for GUI event

sequences and their related callback sequences. Based on this modeling of Android GUI control-flow paths, and the conditions for GPS listener leaks along such paths, we define two patterns of run-time energy-drain behaviors. Next, we develop a static detection algorithm targeting these patterns. The analysis considers valid interprocedural control-flow paths in a callback method and its transitive callees, in order to detect operations that add or remove GPS listeners. Sequences of callbacks are then analyzed for possible GPS listener leaks. Our evaluation considers the detection of GUI-related energy-drain defects reported in prior work, as well as new defects not discovered by prior approaches. The results from this evaluation demonstrate that the detection is very effective and precise, suggesting that the proposed analysis is suitable for practical use in static checking tools for Android.

As the second contribution of this dissertation, we extend the static detection of missing deactivation to Android sensor resources. Similarly to GPS resources, a sensor resource held by an application will not be released by the Android system even if it has been idle for a long time, which will cause battery drain. To detect sensor leaks statically in Android apps, we first perform static modeling of sensor-related objects and API calls. This information is integrated into a static graph model. Graph edges are labeled with symbols representing the opening/closing of UI windows and the acquiring/releasing of sensors. We then define a context-free-language reachability (CFL-R) problem over the graph. A CFL-R graph path is a "witness" of a sensor leak. Given this formulation, we describe an approach to identify leaking paths. The reported paths are then used to generate test cases. The execution of each test case tracks the run-time behavior of sensors and reports observed leaks. Our experimental

results indicate that this approach effectively detects sensor leaks, while focusing the testing efforts on a very small subset of possible GUI event sequences.

Another aspect of energy inefficiency comes from improper usage of energy-intensive Android system services. One such service is the AlarmManager service, which allows an application to execute tasks at specific times. This service is commonly used to perform periodic background tasks such as updating application contents and uploading user data. In normal conditions, these background tasks frequently wake the device from low power sleep. Recent Android releases introduce JobService, a new component that is better suited for such background tasks. While alarms scheduled by AlarmManager are fired at specific times set up by the application, a job defined by a JobService will be batched with other jobs to reduce device wake ups and save energy. As our third contribution, we propose a static analysis to detect energy-inefficient uses of AlarmManager. We define a static reference analysis to determine possible values of parameters used in each alarm scheduling call, which are then used to determine whether the alarm may have negative energy impact. The identified problematic alarms are processed by our automated code refactoring engine and are converted to jobs, while still retaining their original functionality. Our evaluation shows that this approach successfully reduces energy consumption.

In conclusion, this dissertation presents several static analyses to uncover different types of energy-inefficiency patterns of Android applications, with automated testing to verify certain inefficiencies and automated code refactoring to improve energy usage. Our evaluation shows that these analyses are effective and efficient.

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Atanas (Nasko) Rountev for his support, guidance and patience during my Ph.D study. I would also like to thank all current and former PRESTO group for all the collaborations and insightful discussions. I thank Prof. Michael D. Bond and Prof. Neelam Soundarajan for serving on the dissertation committee. I am grateful to Raluca Sauciuc and Sean Klein for their mentoring during my internships at Google. Finally, I would like to thank my parents for their unconditional support.

The material presented in this dissertation is based upon work supported by the U.S. National Science Foundation under grants CCF-1319695 and CCF-1526459, and by a Google Faculty Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VITA

September 2013 – present	Graduate Research Associate, The Ohio State University
Dec 2016	M.S. Computer Science and Engineering, The Ohio State University
June 2013	B.Eng. Information Security, Huazhong University of Science and Technology

PUBLICATIONS

Research Publications

Hailong Zhang, Haowei Wu, and Atanas Rountev. Detection of Energy Inefficiencies in Android Wear Watch Faces. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*, November 2018.

Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static Window Transition Graphs for Android. In *International Journal of Automated Software Engineering (JASE)*, June 2018.

Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-Resilient Library Detection for Android. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'18)*, May 2018.

Haowei Wu, Yan Wang and Atanas Rountev. Sentinel: Generating GUI Tests for Android Sensor Leaks. In *IEEE/ACM International Workshop on Automation of Software Test (AST'18)*, May 2018.

Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated Test Generation for Detection of Leaks in Android Applications. In *IEEE/ACM International Workshop on Automation of Software Test (AST'16)*, May 2016.

Haowei Wu, Shengqian Yang, and Atanas Rountev. Static Detection of Energy Defect Patterns in Android Applications. In *International Conference on Compiler Construction (CC'16)*, March 2016.

Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static Window Transition Graphs for Android. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, November 2015.

Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *International Conference on Software Engineering (ICSE'15)*, May 2015.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Programming Language and Software Engineering	Prof. Atanas Rountev
High-Performance Computing	Prof. P. Sadayappan
Databases/Analytics	Prof. S. Parthasarathy

TABLE OF CONTENTS

Page

Abstract ii

Dedication v

Acknowledgments vi

Vita vii

List of Figures xii

List of Tables xiv

Chapters:

1. Introduction 1

    1.1 Challenges for Detection of Energy-Inefficiency Patterns 1
    1.2 Analysis to Uncover Energy-Inefficiency Patterns 2
        1.2.1 Modeling the Control Flow of Android Applications 2
        1.2.2 Exposing Leaking Behaviors of GPS Listeners 3
        1.2.3 Exposing Leaking Behaviors of Sensor Listeners 5
        1.2.4 Exposing and Correcting Energy-Inefficient Periodic Tasks . 6
    1.3 Outline 7

2. Static Detection of Location-Related Energy Defects 8

    2.1 Background on Android GUIs and Location Listeners 8
        2.1.1 Android GUIs 8
        2.1.2 Adding and Removing Location Listeners 15
    2.2 Patterns of Location-Related Energy Defects 17
        2.2.1 Pattern 1: Lifetime Containment 19

2.2.2 Pattern 2: Long-Wait State . . . 21
2.3 Static Leak Detection for Location Listeners . . . 24
2.3.1 Phase 1: Add-Listener and Remove-Listener Operations . . . 25
2.3.2 Phase 2: Path Generation . . . 28
2.3.3 Phase 3: Detection of Leaking Callback Sequences . . . 31
2.4 Evaluation . . . 33
2.5 Conclusions . . . 39

3. Static Detection of Sensor-Related Energy Defects . . . 40

3.1 Android GUIs and Sensors . . . 41
3.1.1 Android GUI Control Flow . . . 41
3.1.2 Sensors in Android Apps . . . 41
3.2 Control-Flow Analysis for Sensor Leaks . . . 44
3.2.1 Control-Flow Model . . . 44
3.2.2 CFL-Reachability for Sensor Leaks . . . 46
3.2.3 Detection and Reporting of Leaks . . . 50
3.2.4 Generation of Test Cases . . . 52
3.2.5 Static Sensor-Related Abstractions . . . 54
3.3 Analysis Implementation . . . 56
3.4 Evaluation and Case Studies . . . 60
3.5 Case Studies . . . 63
3.6 Conclusions . . . 67

4. Refactoring of Energy-Inefficient Scheduled Tasks . . . 68

4.1 Background . . . 69
4.1.1 Relevant Features of Android's AlarmManager . . . 69
4.1.2 AlarmManager . . . 70
4.1.3 Energy Impact . . . 72
4.1.4 Android JobService . . . 74
4.1.5 Energy-Inefficient AlarmManager Patterns . . . 77
4.2 Semantics of Relevant Android Constructs . . . 78
4.2.1 Plain Java and Plain Android . . . 78
4.2.2 Semantics of AlarmManager . . . 79
4.3 Static Reference Analysis . . . 83
4.3.1 Constraint Graph . . . 84
4.3.2 Constraint Analysis . . . 85
4.3.3 Analysis Output . . . 87
4.4 Automated Code Refactoring . . . 88
4.4.1 Instrument PendingIntent Creation . . . 88
4.4.2 Refactoring the Scheduling of Alarms . . . 89

4.4.3 Construction of JobService for an Alarm 91
4.5 Evaluation 92
4.6 Conclusions 96

5. Related Work 97

5.1 Energy Analysis for Android 97
5.2 Test Generation for Android 100
5.3 Automated Refactoring for Android 101
5.4 Static Control-Flow Analysis for Android 101

6. Conclusions 103

BIBLIOGRAPHY 106

LIST OF FIGURES

Figure **Page**

2.1 Notation for run-time semantics. 9

2.2 Example derived from the DroidAR application. 11

2.3 Notation for listener semantics. 17

2.4 Example derived from the Ushahidi application. 22

2.5 WTGs for the running examples. 26

3.1 Example derived from Calculator Vault. 42

3.2 SG graph for the running example. 45

3.3 Finite automaton $\mathcal{F}$. 47

3.4 Example derived from CSipSimple. 49

3.5 Pushdown automaton $\mathcal{P}$. 51

3.6 Example of generated test case. 53

3.7 Example derived from Geopaparazzi. 65

4.1 Example derived from the Moloko application. 71

4.2 Illustration of different wake up patterns 73

4.3 Example derived from the Muzei application. 75

4.4 Semantic domains and functions 79

4.5 Abstract program representation. 84

4.6 Example of a constraint graph 85

4.7 Recording of created pending intents. 89

4.8 Example of inserted method. 90

4.9 Example of constructed `JobService` for Moloko app. 92

4.10 Battery level comparison 94

4.11 Device wake up comparison 95

LIST OF TABLES

Table **Page**

2.1 Analyzed applications and detected defects. 34

2.2 Defects reported: GreenDroid (D) vs static analysis (S). 37

3.1 Applications, paths, and tests. 61

CHAPTER 1: Introduction

Android has become dominant platform in the smartphone market worldwide. Devices powered by Android are used by increasing numbers of users in their everyday lives, for a variety of tasks such as social networking, online purchases, casual gaming, and playback of audio and video. Since mobile devices have limited CPU power, memory, and battery capacity, software defects related to inefficient use of these resources can have substantial impact on the user experience. In particular, as indicated by other researchers, *energy-related defects* are of significant interest to developers and app users. On an Android device, such defects can be caused by different energy-inefficiency patterns. The focus on this dissertation is the analysis of several such patterns.

1.1 Challenges for Detection of Energy-Inefficiency Patterns

Limitation of dynamic analysis While defects caused by energy inefficiency can be observed by measuring the energy consumption of the running application, such measurements are not always easy to perform. Furthermore, it is difficult to track down the root causes of observed defects. Thus, dynamic analysis techniques require significant effort. Developers typically start with initial test cases and use profilers to identify the code areas that consume large portion of resources. However, for resources that consume energy (e.g., GPS and hardware sensors), there are no reliable ways to associate run-time energy consumption with profiler's results. Furthermore, it is

difficult to perform this type of measurements without specialized hardware. Even if the profiler can locate code regions where energy consumption is high, significant human effort is needed to identify correct fixes. Moreover, if a defective code region is not covered by developer-written test cases, the profiler cannot discover it.

Limitation of static analysis Compared to dynamic analysis, static analysis examines code paths in the application without actually executing them. Since Android applications are event-driven, run-time app behavior is based on a sequence of callbacks from the platform code to the application code. This behavior is not modeled fully by existing static analyses, which could cause false negatives because unmodeled event-driven behavior leads to under-approximations of possible run-time behaviors. Furthermore, the modeling of control-flow in existing work is also over-approximating some aspects of the run-time behavior. This may introduce infeasible code paths that could cause false positives.

1.2 Analysis to Uncover Energy-Inefficiency Patterns

The goal of this dissertation is to develop several static program analyses to uncover energy-inefficiency patterns in Android applications. For certain patterns, we also perform automated test generation to verify detected defects, as well as automated code refactoring to reduce energy consumption.

1.2.1 Modeling the Control Flow of Android Applications

Unlike a traditional program that usually has a fixed entry point (e.g., the main function in a C program), the code of an Android app can be entered in multiple ways. Because Android applications are GUI-based and event-driven, they contain large numbers of callback methods. These methods are invoked from the framework

when different types of events are sent to the application. Depending on the context of a single event, the triggered callback methods can be significantly different, which makes the control flow analysis even harder. The modeling of Android GUIs is also essential for precise control flow analysis, since the relevant events are triggered on GUI elements. In Android, the widgets of a screen window can be constructed from a predefined layout definition and/or dynamically changed through several APIs at run time, which makes GUI modeling a challenging problem. In this dissertation, we adapt the modeling of Android GUIs from prior work [93, 94] as the basis of our analyses. This work models the relationships among windows (Android activities, dialogs, etc.), widgets (Android views), and their event handlers. This information is then used to construct the so-called window transition graph (WTG), which represents the run-time transitions between GUI windows caused by events. The details of this model are explained in Chapter 2. This representation is the basis of our static analyses of energy-inefficiency patterns.

1.2.2 Exposing Leaking Behaviors of GPS Listeners

For mobile devices, the management of energy-intensive resources (e.g., GPS) burdens the developer with "power-encumbered programming" [69] and creates various opportunities for software defects. Static detection of such defects is of significant value. Common battery-drain defects—"no-sleep" [69] and "missing deactivation" [10, 55]—are due to executions along which *an energy-draining resource is activated but not properly deactivated*. Such dynamic behaviors can be naturally stated as properties of control-flow paths, and thus present desirable targets for static control-flow and data-flow analyses. We aim to develop a general static analysis approach for

detecting certain common categories of energy-drain defects. Specifically, we aim to detect "missing deactivation" behaviors in the user interface thread of the application.

The proposed approach is based on three key contributions. First, we define precisely two patterns of run-time energy-drain behaviors. The definition is based on formal definitions of relevant aspects of Android GUI run-time control flow, including modeling of GUI events, event handlers, transitions between windows, and the associated sequences of callbacks. This modeling allows us to define the notion of a leaking control-flow path and two defect patterns based on it. These patterns are related to the (mis)use of Android location awareness capabilities (e.g., GPS location information). Location awareness is a major contributor to energy drain [28] and in prior work on dynamic defect detection [55] has been identified as the predominant cause of energy-related defects in UI behavior. Our definition of defect patterns is motivated by case studies from this prior work and by our own analysis of these case studies. However, our careful formulation of these patterns is new and provides a valuable contribution to the state of the art. Furthermore, our control-flow modeling is significantly more general than any prior technique. The second contribution of our approach is a static defect detection algorithm (Section 2.3). Based on the WTG model, the analysis considers valid interprocedural control-flow paths in each callback method and its transitive callees, in order to detect operations that add or remove location listeners. Sequences of window transitions and their callbacks are then analyzed for possible listener leaking behaviors based on the two patterns mentioned earlier. The third contribution is a study of the effectiveness of the proposed static detection (Section 2.4). We aim to determine how well the analysis discovers GUI-related energy-drain defects reported in prior work, as well as new defects not

discovered by prior approaches. Our evaluation indicates that the static detection is very effective and is superior to dynamic detection. Furthermore, all but one of the reported problems are real defects. The evaluation also shows that the cost of the analysis is low. This high precision and low cost suggest that the proposed approach is suitable for practical use in static checking tools for Android.

This work first appeared at the International Conference on Compiler Construction [85].

1.2.3 Exposing Leaking Behaviors of Sensor Listeners

Hardware sensors on mobile devices are used to sense environment changes in acceleration, rotation, luminance, etc. Sensors provide opportunities for Android developers to offer rich app functionality. However, the use of sensors creates opportunities for energy inefficiencies. As a general Android developer guideline, the app should always disable sensors that are not needed. Failing to disable unneeded sensors—that is, *sensor leaks*—can drain the battery. If possible, sensor leaks should be detected and eliminated before an app is released in an app store.

The second contribution of this dissertation is a static analysis, combined with a test generation approach, to detect such sensor leaks. We first perform static modeling of sensor-related objects and API calls. This information is integrated into a static graph model. Graph edges are labeled with symbols representing the opening/closing of UI windows and the acquiring/releasing of sensors. We then define two sensor leaks patterns, described in Section 3.2.2. A context-free-language reachability (CFL-R) is defined for these patterns over the graph. We next describe an approach to identify and report buggy paths. This approach traverses selected CFL-R paths and

checks them for potential leaks. Reported paths are then used to generate test cases. The execution of each test case tracks the run time behavior of sensors and reports observed sensor leaks. The evaluation results, presented in Section 3.4, indicate that our approach effectively detects sensor leaks, while focusing the testing efforts on a very small subset of possible GUI event sequences, as determined by our targeted sensor-aware static analysis of app code.

An earlier version of this work appeared at the IEEE/ACM International Workshop on Automation of Software Test [86].

1.2.4 Exposing and Correcting Energy-Inefficient Periodic Tasks

It is a common practice that Android applications run periodic background tasks by scheduling alarms using the `AlarmManager` system service. When multiple applications use `AlarmManager`, this may cause frequent device wake ups, which impacts the device's energy consumption. Later Android versions offer better services (e.g., `JobService`) to execute periodic tasks in an energy-efficient manner. In Chapter 4, we propose a static analysis to uncover such energy inefficiencies due to the use of `AlarmManager`. We then combine this analysis with automated code instrumentation and refactoring, in order to convert the problematic code to `JobService`, which allows for these periodic tasks to be batched together to reduce CPU wake ups. We evaluate this approach by measuring the energy consumption before and after the code refactoring. Our results, presented in Section 4.5, show that this novel approach for analysis and code transformation successfully reduces energy consumption.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 describes the program analysis that detects energy defects due to leaks of GPS listeners. Chapter 3 presents the program analysis and test generation to uncover energy defects due to sensor-related leaks. Chapter 4 defines the program analysis and automated code refactoring to detect and transform improper uses of the `AlarmManager` service. Chapter 5 describes related works and Chapter 6 summarizes the contributions of this dissertation.

CHAPTER 2: Static Detection of Location-Related Energy Defects

On an Android device, GPS hardware provides location information to the system software and applications, which allows the developer to build location-aware applications. While this feature provides convenience to the users and developers, it can also cause battery drain issues if not handled correctly. For example, if a location-aware application registers a location listener (an object that contains callback methods that will be invoked when there is an update from the GPS) and still keeps it when the application is put into the background, the device battery will be exhausted in a short period of time. In this chapter, we target the energy defects caused by "missing deactivation" of GPS location listeners. Section 2.1 explains the run-time semantics of Android GUIs and location listeners. Section 2.2 defines the targeted energy defects. Section 2.3 introduces the static analysis algorithms used to detect these defects. Section 2.4 provides the evaluation of our static analysis.

2.1 Background on Android GUIs and Location Listeners

2.1.1 Android GUIs

We start with an overview of Android GUI run-time semantics. This is needed because the patterns of location-related energy defects we target can be defined based on the sequence of GUI actions and states. Figure 2.1 summarizes our notation for these features. In this work, we focus on the event-driven control flow in the GUI

$w \in$	**Win**	window
$v \in$	**View**	view
$e = [v,k] \in$	**Event**	widget event on v
$e = [w,k] \in$	**Event**	default event on w
$c \in$	**Cb**	callback method
$[c,o] \in$	$\mathbf{Cb} \times (\mathbf{Win} \cup \mathbf{View})$	callback invocation
$s \in$	**Cbs**	callback inv. sequence
$t = [w,w'] \in$	**Trans**	window transition
$\epsilon(t) \in$	**Event**	event that triggered t
$\sigma(t) \in$	**Cbs**	callback sequence for t
$\delta(t) \in$	$(\{push, pop\} \times \mathbf{Win})^*$	window stack changes
$T \in$	$\mathbf{Trans}^+$	transition sequence

Figure 2.1: Notation for run-time semantics.

of the application (i.e. in the main application thread). Figure 2.2 and Figure 2.4 contain two code examples to illustrate these features.

Windows and views *Activities* are core components of Android applications, defined by subclasses of `android.app.Activity`. An activity displays a window containing several GUI widgets. A widget (a "view" in Android terminology) is an object from a subclass of `android.view.View`.

▷ *Example:* Figure 2.2 shows an example derived from an energy-drain defect we found in the DroidAR application analyzed in prior work on the GreenDroid dynamic defect detection tool [55]. This particular defect is new, and was not reported in that prior work. The defect is due to the lack of a listener-remove operation at the end of the lifetime of activity `DemoLauncher`. When button `btnRun` is clicked, the `onClick` handler registers a location listener (at line 16) but this listener is not removed even

after the activity is destroyed. As a result, location data (e.g., GPS reads) is sensed even after it is not needed anymore, which will drain the battery.

Class `DemoLauncher` is an example of an activity. In this case this is the start activity of the application: it is started by the Android launcher when the user launches the application. The `onResume` lifecycle callback (discussed shortly) retrieves a button widget `btnRun` at line 3. This widget is then associated with an event handler callback `onClick`, defined in an anonymous class that implements interface `OnClickListener` (lines 4–5). Parameter `v` in this callback refers to the button widget. ◁

We also consider the two other common categories of Android windows: menus and dialogs. Instances of menu classes represent short-lived windows associated with activities ("options" menus) and widgets ("context" menus). An example presented later in this chapter illustrates the use of menus. A dialog is an object from some subclass of `android.app.Dialog`. Both menus and dialogs require users to take an action before they can proceed [27]. A menu/dialog implements a simple interaction with the user, and its lifetime is shorter than activity lifetime. The last activity that was displayed before a menu/dialog was displayed is the *owner activity* of this menu/dialog. The lifetime of a menu or a dialog is contained within the lifetime of its owner activity.

We will use **Win** to denote the set of all run-time windows (activities, menus, and dialogs) and **View** for the set of all run-time widgets in these windows (Figure 2.1).

Events Each $w \in$ **Win** can respond to several events. *Widget events* are of the form $e = [v,k]$ where $v \in$ **View** is a widget and k is an event kind. For the example

```
1  class DemoLauncher extends Activity {
2    void onResume() {
3      Button btnRun = ...;
4      btnRun.setOnClickListener(new OnClickListener() {
5        void onClick(View v) { run(); } }); }
6    // === Other possible lifecycle callbacks: onCreate,
7    // === onDestroy, onStart, onRestart, onPause
8    void run() {
9      EventManager.getInstance().registerListeners(); } }
10
11 class EventManager implements LocationListener {
12   static EventManager instance = new EventManager();
13   static getInstance() { return instance; }
14   void registerListeners() {
15     LocationManager lm = ...;
16     lm.requestLocationUpdates(this); } }
```

Figure 2.2: Example derived from the DroidAR application.

in Figure 2.2 we have event $[br,click]$ where br is the `Button` instance referenced by `btnRun`.

We also consider five kinds of *default events*. Event *back* corresponds to pressing the hardware BACK button, which closes the current window w and typically (but not always) returns to the window that opened w.[1] Event *rotate* shows that the user rotates the screen, which triggers various GUI changes. Event *home* abstracts a scenario there the user switches to another application and then resumes the current application (e.g., by pressing the hardware HOME button to switch to the Android application launcher, and then eventually returning back to the application). Event *power* represents a scenario where the device screen is turned off by pressing the

[1]In some scenarios (e.g., callback `onBackPressed` is defined) the window is not closed. We have not observed such scenarios in the analyzed apps.

hardware POWER button, followed by device reactivation. Event *menu* shows the pressing of the hardware MENU button to display an options menu. A default event will be represented as $e = [w,t] \in \mathbf{Win} \times \{back, rotate, home, power, menu\}$ where w is the currently-active window. We will use **Event** to denote the set of all widget events and default events.

In Figure 2.2 we have five default events [`DemoLauncher`,. . .], but since the activity does not define an options menu, event *menu* does not have any effect.

Callbacks Each $e \in \mathbf{Event}$ triggers a sequence of *callback invocations* that can be abstracted as $[c_1,o_1][c_2,o_2]\ldots[c_m,o_m]$. Here c_i is a callback method defined by the application, and o_i is a run-time object on which c_i was triggered. Note that each of these invocations completes before the next one starts—that is, their lifetimes are not nested within each other, but rather they are disjoint. The actual invocations are performed by event-processing logic implemented inside the Android framework code.

We consider two categories of callbacks. *Widget event handler callbacks* respond to widget events; an example is `onClick` in Figure 2.2. *Lifecycle callbacks* are used for lifetime management of windows. For example, creation callback `onCreate` indicates the start of the activity's lifetime, and termination callback `onDestroy` indicates end of lifetime. Menus and dialogs can also have create/terminate callbacks.

▷ *Example:* In Figure 2.2 event $[br,click]$ (br is the `Button` object referenced by `btnRun`) will cause a widget event handler callback invocation [`onClick`,br]. In this example the callback sequence contains only this invocation. However, for the sake of the example, suppose that `onClick` invoked an Android API call to start some new

activity a. Also, for illustration, suppose that the source activity `DemoLauncher` and the target activity a both define the full range of activity lifecycle callbacks (listed for completeness at lines 6–7 in Figure 2.2). Then the callback invocation sequence would be [`onClick`,br][`onPause`,`DemoLauncher`][`onCreate`,a][`onStart`,a][`onResume`,a] [`onStop`,`DemoLauncher`]; this sequence can be observed via `android.os.Debug` tracing.

If after [br,$click$] the next event was [a,$back$]—that is, the BACK button was pressed to close a and return to `DemoLauncher`—the sequence would be [`onPause`,a] [`onRestart`,`DemoLauncher`] [`onStart`,`DemoLauncher`][`onResume`,`DemoLauncher`] [`onStop`, a] [`onDestroy`,a]. As seen from these examples, there can be a non-trivial sequence of callback invocations in response to a single GUI event. ◁

Window transitions We use the term *run-time window transition* to denote a pair $t = [w,w'] \in \textbf{Win} \times \textbf{Win}$ showing that when window w was active and interacting with the user, a GUI event occurred that caused the new active window to be w' (w' may be the same as w). Each transition t is associated with the event $\epsilon(t) \in \textbf{Event}$ that caused the transition and with $\sigma(t)$, a sequence of callback invocations $[c_i,o_i]$.

There are two categories of callback invocation sequences for Android GUI transitions. The first case is when event $\epsilon(t)$ is a widget event $[v,k]$ where v is a widget in the currently-active window w. In this case $\sigma(t)$ starts with $[c_1,v]$ where c_1 is the callback responsible for handling events of type k on v. The rest of the sequence contains $[c_i,w_i]$ with c_i being a lifecycle callback on some window w_i. In general, the windows w_i whose lifecycles are affected include the source window w, the target window w', as well as other related windows (e.g., the owner activity of w). In the

running example, a self-transition t for `DemoLauncher` is triggered by event [br,$click$], resulting in $\sigma(t)$ = [`onClick`,br]. Following the hypothetical example from above, if `onClick` opens another activity a, the transition would be from `DemoLauncher` to a, with $\sigma(t)$ as listed above: [`onClick`,br] ... [`onStop`,`DemoLauncher`].

The second category of callback sequences is when $\epsilon(t)$ is a default event [w,k] on the current window w. In this case all elements of $\sigma(t)$ involve lifecycle callbacks. For example, event *home* on `DemoLauncher` triggers a self-transition t with $\sigma(t)$ containing invocations of `onPause`, `onStop`, `onRestart`, `onStart`, `onResume` on that activity. Additional details of the structure of these callback sequences are presented in our earlier work [91, 93, 94].

Window stack Each transition t may open new windows and/or close existing ones. This behavior can be modeled with a *window stack*: the stack of currently-active windows.[2] Each transition t can modify the stack by performing window push/pop sequences. These effects will be denoted by $\delta(t) \in (\{push, pop\} \times \mathbf{Win})^*$. In the examples presented in this chapter, the effects of a transition t are relatively simple: for example, opening a new window w represented by *push* w, or closing the current window w represented by *pop* w. In the simplest case, as in the self-transition t from Figure 2.2, $\delta(t)$ is empty. However, our prior work [93, 94] shows that in general these effects are more complex: $\delta(t)$ could be a (possibly empty) sequence of window pop operations, followed by an optional push operation. These operations could involve several windows and can trigger complicated callback sequences.

[2]Features such as launch modes for activities [25] can lead non-LIFO behaviors, but they do not appear to be commonly used [89].

Transition sequences Consider any sequence of transitions $T = \langle t_1, t_2, \ldots, t_n \rangle$ such that the target of t_i is the same as the source of t_{i+1}. Let $\sigma(T)$ be the concatenation of callback sequences $\sigma(t_i)$; similarly, let $\delta(T)$ be the concatenation of window stack update sequences $\delta(t_i)$. Sequence T is *valid* if $\delta(T)$ is a string in a standard context-free language [73] defined by

$$Valid \rightarrow Balanced\ \ Valid \mid push\ w_i\ \ Valid \mid \epsilon$$

where *Balanced* describes balanced sequences of matching push and pop operations

$$Balanced \rightarrow Balanced\ \ Balanced \mid push\ w_i\ \ Balanced\ \ pop\ w_i \mid \epsilon$$

2.1.2 Adding and Removing Location Listeners

The callbacks invoked during window transitions can perform a variety of actions. Our work considers actions that may affect energy consumption. In particular, we focus on *add-listener* and *remove-listener* operations related to location awareness. Such actions have been considered by GreenDroid [55], an existing dynamic analysis tool for detection of energy defects in Android applications. Since almost all GUI-related energy-drain defects reported in this prior work are due to location awareness, focusing on such defects allows us to perform direct comparison with the results from this existing study.

Relevant Android APIs The standard mechanism for obtaining information about the location of the user (e.g., using GPS data) is by registering a *location listener* with the framework's location manager. The listener implements callback methods that are invoked when relevant changes happen. Registration is done through API calls

such as `requestLocationUpdates`, with the location listener provided as a parameter. The listener can be removed by calling `removeUpdates` and using the listener as a parameter. A number of other Android APIs have similar effects. For example, when an application is displaying a map, it could display an overlay of the current user location on the map by calling `enableMyLocation` on an overlay object, in which case this object becomes a listener for location updates. A subsequent call to `disableMyLocation` stops this listening.

▷ *Example:* In Figure 2.2 callback [`onClick`,*br*] invokes `run`, which in turn invokes `registerListeners` on an instance of `EventManager`. This instance (created at line 12) is a listener object that is registered for location updates at line 16. However, this listener is never removed by any other code in the activity. In particular, if the user exits `DemoLauncher`—e.g., by pressing the hardware BACK button to exit the application—the listener will remain registered and will drain the battery. We have confirmed this incorrect behavior through testing. ◁

The standard guidelines for building location-aware applications warn the developers to *"always beware that listening for a long time consumes a lot of battery power"* [28]. Our goal is to model the addition and removal of location listeners along sequences of window transitions (and their related callbacks), in order to identify problematic behaviors that may lead to extended periods of location listening. We formalize the relevant run-time features as follows (also see Figure 2.3). Let **Lst** be the set of all run-time objects l that are location listeners. Let **Op** be the set of pairs $[a,r]$ where a is an API method for adding a listener and r is the corresponding API method for removing that listener. We will use $\mathbf{Op}_a$ to denote $\{a \mid [a,r] \in \mathbf{Op}\}$; $\mathbf{Op}_r$

$w \in$	**Win**	window
$v \in$	**View**	view
$e = [v,k] \in$	**Event**	widget event on v
$e = [w,k] \in$	**Event**	default event on w
$c \in$	**Cb**	callback method
$[c,o] \in$	$\mathbf{Cb} \times (\mathbf{Win} \cup \mathbf{View})$	callback invocation
$s \in$	**Cbs**	callback inv. sequence
$t = [w,w'] \in$	**Trans**	window transition
$\epsilon(t) \in$	**Event**	event that triggered t
$\sigma(t) \in$	**Cbs**	callback sequence for t
$\delta(t) \in$	$(\{push, pop\} \times \mathbf{Win})^*$	window stack changes
$T \in$	$\mathbf{Trans}^+$	transition sequence
$l \in$	**Lst**	listener
$op = [a,r] \in$	**Op**	add/remove listener APIs
$A \subseteq$	$\mathbf{Op}_a \times \mathbf{Lst}$	added listeners
$R \subseteq$	$\mathbf{Op}_r \times \mathbf{Lst}$	removed listeners

Figure 2.3: Notation for listener semantics.

is defined similarly. For example, [`requestLocationUpdates`,`removeUpdates`] $\in$ **Op**. The extended notation based on these features is shown in Figure 2.3.

2.2 Patterns of Location-Related Energy Defects

Leaking sequences Consider $s = [c_1,o_1][c_2,o_2]\ldots[c_m,o_m]$, a callback sequence observed during some window transitions. Recall that c_i is a callback method and o_i is a view/window on which c_i is called. Let A_i be the set of pairs $[a,l] \in \mathbf{Op}_a \times \mathbf{Lst}$ such that add-listener method a was invoked on listener l during the execution of c_i on o_i, and the rest of the execution of c_i did not invoke r on l for any $[a,r] \in \mathbf{Op}$. In other words, c_i (or its transitive callees) invoked a and provided l as a parameter, and subsequently did not invoke on l any remove-listener method r that matches a. One can

draw an analogy with the standard compiler notion [2] of a downward-exposed definition in a basic block (i.e., a definition that reaches the exit of the block). Similarly, we will use *downward-exposed* to denote any $[a,l]$ that was not killed by a subsequent $[r,l]$ in c_i and its transitive callees, and thus reached the exit of c_i.

In the running example, for the callback sequence containing only [`onClick`,br], the corresponding set A_1 contains one element: [`requestLocationUpdates`,l] where l is created at line 12.

Similarly, for callback invocation $[c_i,o_i]$, let R_i contain $[r,l] \in \mathbf{Op}_r \times \mathbf{Lst}$ such that remove-listener method r was invoked on l during the callback execution. Note that the definition of R_i could have included the following additional condition: "in the execution of c_i (and its transitive callees) $[r,l]$ was not preceded by a matching $[a,l]$". Such a condition would have made the definition of R_i similar to the definition of A_i. However, such a condition is not necessary because in Android it is possible for a single $[r,l]$ to be preceded by several matching $[a,l]$, occurring over multiple callbacks, including the callback c_i that invokes $[r,l]$. That single remove operation "cancels" all preceding add operations. Thus, it is irrelevant whether c_i contains a preceding $[a,l]$.

Definition 1 *Given a callback invocation sequence s of length m, let $A_1, A_2, \ldots, A_m$ be its sequence of add-listener sets and $R_1, R_2, \ldots, R_m$ be its sequence of remove-listener sets. Sequence s leaks listener l if there exists an add-listener operation $[a,l] \in A_i$ such that for each $j > i$ there does not exist a matching $[r,l] \in R_j$.*

Leaking callback sequences are typically harmless: they represent legitimate needs to receive updated information about the location of the device. For example, in

Figure 2.2, the click event on the button causes a window self-transition with a leaking callback sequence (containing only [`onClick`,br]), but of course this is the intended behavior. However, not all leaking sequences are desired. We define two patterns of leaking sequences that represent potential defects. These patterns are motivated by case studies from the work on GreenDroid, and by our own analysis of these case studies.

2.2.1 Pattern 1: Lifetime Containment

The informal definition of this pattern is as follows: if an activity adds a listener, the listener should be removed before that activity is destroyed. A similar pattern has been defined informally as part of the GreenDroid tool, but our goal here is to state it precisely in order to allow the development of static detection algorithms. Note that since menus and dialogs are intended to be short-lived, their lifetimes cannot be expected to contain the lifetime of listener registration; thus, the pattern is defined only for activities.

Consider a window transition sequence $T = \langle t_1, t_2, \ldots, t_n \rangle$ and recall that $\delta(T)$ is the concatenation of window stack push/pop operations $\delta(t_i)$. Sequence T represents a lifetime of an activity w if $\delta(t_1)$ contains an operation *push* w, $\delta(t_n)$ contains an operation *pop* w, and the sequence of push/pop operations between these two operations in $\delta(T)$ is balanced (as defined by non-terminal *Balanced* described earlier).

Definition 2 *Suppose T represents a lifetime of an activity w. Consider the callback invocation sequence $\sigma(T)$ and its subsequence $s = [c_1,w] \ldots [c_m,w]$ where c_1 is the creation callback for w and c_m is the termination callback for w. If s is leaking a*

listener l, and the corresponding add-listener operation $[a,l]$ occurred when the top-most activity on the window stack was w, then T matches Pattern 1.

Note that this definition does not say "the top element on the window stack was w". Here we allow for some menus/dialogs (owned by w) to be on top of w in the window stack at the time when the listener is added. Since menus and dialogs often execute small actions on behalf of their owner activity, we still attribute the add-listener operation to the activity, and consider whether the activity's lifetime contains the lifetime of listener registration.

▷ *Example:* Consider `DemoLauncher` in the running example. For brevity, let us denote it with a. Since this is the starting activity of the application, we introduce an artificial transition $t_1 = [launcher,a]$ where *launcher* denotes the Android application launcher. The relevant callbacks are $\sigma(t_1) = [\texttt{onCreate},a][\texttt{onStart},a][\texttt{onResume},a]$. Next, let $t_2 = [a,a]$ be the transition triggered by button br, with $\sigma(t_2) = [\texttt{onClick},br]$. Finally, let $t_3 = [a,launcher]$ occur then the hardware BACK button is pressed to close a and go back to the Android launcher. The callbacks are $\sigma(t_3) = [\texttt{onPause},a]$ $[\texttt{onStop},a][\texttt{onDestroy},a]$. Let window transition sequence $T = \langle t_1, t_2, t_3 \rangle$. This sequence is balanced, since the window stack effects are $\delta(t_1) = push\ a$, $\delta(t_2) = [\ ]$, and $\delta(t_3) = pop\ a$. Clearly, T represents a lifetime of a. In the application code, the activity defines only `onCreate` (not shown in the figure because it does not have relevant effects) and `onResume`, but not any other lifecycle callbacks. Thus, the relevant callback sequence is $[\texttt{onCreate},a][\texttt{onResume},a][\texttt{onClick},br]$. Since the last callback contains $[\texttt{requestLocationUpdates},l]$ and there is no subsequent $[\texttt{removeUpdates},l]$, window transition sequence T matches Pattern 1. ◁

2.2.2 Pattern 2: Long-Wait State

Informally, this pattern considers an activity that adds a listener and then is put in a (potentially) long-wait state without removing this listener. We are interested in application states that can suspend the application for arbitrarily long periods of time. Specifically, suppose the the application user presses the hardware HOME button (or, equivalently, selects another application from the list of recent applications). As a result, the current application is put in the background. However, if there are any active location listeners, the battery's energy is still consumed. It may be hours before the user resumes the application. A similar scenario occurs when the hardware POWER button is pressed: the screen is turned off, but active location listeners still drain the battery. Since the callback sequences for these two scenarios are the same, we will discuss only the use of the HOME button to put the application in a long-wait state. This pattern has not been identified in the work on GreenDroid.

Unlike with Pattern 1, here the lifetime of the activity may contain the lifetime of listener registration. This scenario is illustrated by the example in Figure 2.4. The simplified code in the figure is derived from the Ushahidi application, which was also analyzed in the prior work on GreenDroid. This particular defect is not detected in the experiments from that prior work. Through testing, we have confirmed that indeed this defect drains the battery.

▷ *Example:* Activity `ListCheckin` contains an options menu. Event handler `onOptionsItemSelected` represents the clicking of an item in that menu. One of the menu items is used to open `CheckinActivity`, using the standard Android mechanism of intents. Inside the newly-opened activity, `onCreate` registers the activity as a listener, and `onDestroy` stops the listening. As far as Pattern 1 is concerned, the

```
1  class ListCheckin extends Activity {
2    void onOptionsItemSelected(MenuItem item) {
3      switch(item.getItemId()) { ...
4        case ADD_INCIDENT:
5          Intent i = new Intent(CheckinActivity.class);
6          startActivity(i); return; ... } }
7
8  class CheckinActivity extends Activity
9        implements LocationListener {
10   LocationManager lm = ...;
11   void onCreate() {
12     lm.requestLocationUpdates(this); }
13   void onDestroy() {
14     lm.removeUpdates(this); }
15   void onStart() { ... }
16   void onResume() { ... }
17   void onPause() { ... }
18   void onLocationChanged() {
19     lm.removeUpdates(this); } }
```

Figure 2.4: Example derived from the Ushahidi application.

lifetime of listener registration is contained within the lifetime of the activity that adds the listener. The rest of the lifecycle callbacks defined in the code (lines 15–17) do not have any effect on the listener.

Callback `onLocationChanged` is invoked when a location read is obtained; this method stops the listening. However, it is still possible for the listener to be leaked. If a location read cannot be acquired (e.g., the GPS cannot obtain a satellite fix because of physical obstacles or atmospheric conditions), callback `onLocationChanged` will not be invoked. If at this moment the user presses the HOME or POWER button (e.g., the user gives up after the GPS signal cannot be acquired), the application is put on the background but the listening is still active and is draining the battery.

Note that battery drain for such "no-read" listener leaks occurs at the same rate as drain for "normal" listener leaks when location reads are successfully acquired. ◁

To define the general form of this defect pattern, consider a window transition sequence $T = \langle t_1, t_2, \ldots, t_n \rangle$ where $\delta(t_1)$ contains *push* w for an activity w and event $\epsilon(t_n)$ is [w',*home*] where w' is either the same as w, or is a menu/dialog owned by w. Here w is created at the beginning of T and default event *home* occurs on w' at the end of T. Suppose also that the sequence of push/pop operations in $\delta(T)$, starting from *push* w on t_1, is valid as defined by non-terminal *Valid* described earlier. Under these conditions, T puts w in a long-wait state.

Definition 3 *Suppose T puts an activity w in a long-wait state. Consider the callback invocation sequence $\sigma(T)$ and its subsequence $s = [c_1,w] \ldots [c_m,w']$ where c_1 is the creation callback for w and c_m is the last callback before the application goes in the background. If s is leaking a listener l, and the corresponding add-listener operation $[a,l]$ occurred when the top-most activity on the window stack was w, then T matches Pattern 2.*

Note that the last callback c_m in this definition is an intermediate point in the invocation sequence for the last transition t_n. If an activity defines all lifecycle callbacks, the entire sequence for t_n is `onPause`, `onStop`, `onRestart`, `onStart`, `onResume`. Callback c_m is `onStop`, since the first two callbacks occur before the application goes in the background, and the last three are executed during reactivation when the user returns to the application.

▷ *Example:* For the example in Figure 2.4, let us use (for brevity) a to denote `ListCheckin`, m to denote the options menu of a, and a' to denote `CheckinActivity`.

Consider the transition sequence $T = \langle t_1, t_2 \rangle$ defined as follows. Transition t_1 is triggered when the `ADD_INCIDENT` menu item is clicked. The window stack effect sequence $\delta(t_1)$ is *pop* m, *push* a'. Here we account for the standard behavior of menus: after a menu item is clicked, the menu is automatically closed—thus, *pop* m should be included in the sequence. Transition t_2 is triggered by event [a',*home*] and has an empty $\delta(t_2)$. Clearly, T represents a valid sequence of transitions.

In general, $\sigma(t_1)$ is [`onOptionsItemSelected`,ai][`onOptionsMenuClosed`,m] [`onPause`,a][`onCreate`,a'][`onStart`,a'][`onResume`,a'][`onStop`,a]; here ai is the menu item with id `ADD_INCIDENT`. Sequence $\sigma(t_2)$ contains [`onPause`,a'][`onStop`,a'][`onRestart`,a'] [`onStart`,a'][`onResume`,a']. The last three callbacks in $\sigma(t_2)$ occur after the application is re-activated from the background. Accounting for the subset of callbacks defined in the code, and the definition of Pattern 2, the relevant callbacks are [`onCreate`,a'] [`onStart`,a'][`onResume`,a'][`onPause`,a']. The first element in the sequence adds the activity as a listener, but the rest of the callbacks do not remove this listener. Thus, $T = \langle t_1, t_2 \rangle$ matches Pattern 2. A fix for the defect would be to remove the listener in `CheckinActivity.onPause`. ◁

2.3 Static Leak Detection for Location Listeners

The run-time behaviors defined earlier can be used as basis for defining static abstractions and static detection analyses based on them. In a minor abuse of notation, for the rest of this chapter we will use **Win**, **View**, etc. (Figure 2.3) to denote sets of static abstractions rather than run-time entities. There are various ways to define such static abstractions. We use the approach from [74, 89], which creates a separate $a \in$ **Win** for each activity class, together with appropriate $m, d \in$ **Win** for its menus

and dialogs, and abstractions $v \in \mathbf{View}$ for their widgets (i.e., defined programmatically or in layout XML files), and then propagates them similarly to interprocedural points-to analysis, but with special handling of Android API calls.

Follow-up work [91–94] defines the *window transition graph* (WTG), a static model $G = (\mathbf{Win}, \mathbf{Trans}, \epsilon, \delta, \sigma)$ with nodes $w \in \mathbf{Win}$ and edges $t = [w,w'] \in \mathbf{Trans}$. Each transition t is annotated with trigger event $\epsilon(t)$, callback sequence $\sigma(t)$, and window stack changes $\delta(t)$. Implementations of these analyses are available in our GATOR [24] analysis toolkit for Android, which itself is built using the Soot framework [83] and its Jimple internal representation (constructed either from Java bytecode or from Dalvik bytecode via Dexpler [13]). The energy defect analysis was developed in this infrastructure.

The WTGs for the running examples are shown in Figure 2.5. These graphs are small because the examples are simplified on purpose, but in actual applications WTGs have hundreds of edges. Given the WTG, our detection analysis proceeds in three phases.

2.3.1 Phase 1: Add-Listener and Remove-Listener Operations

Consider the set $\{[c,o] \mid t \in G \wedge [c,o] \in \sigma(t)\}$. For each invocation of a callback c on object o, we compute a set $A(c,o)$ of pairs $[a,l]$ of an add-listener API invocation statement a and a listener object l. We also compute a similar set $R(c,o)$ of pairs $[r,l]$. These sets are determined in four steps, as described below.

Step 1 An interprocedural control-flow graph (ICFG) [76] is constructed for c and its transitive callees in the application code. Then, a constant propagation analysis is

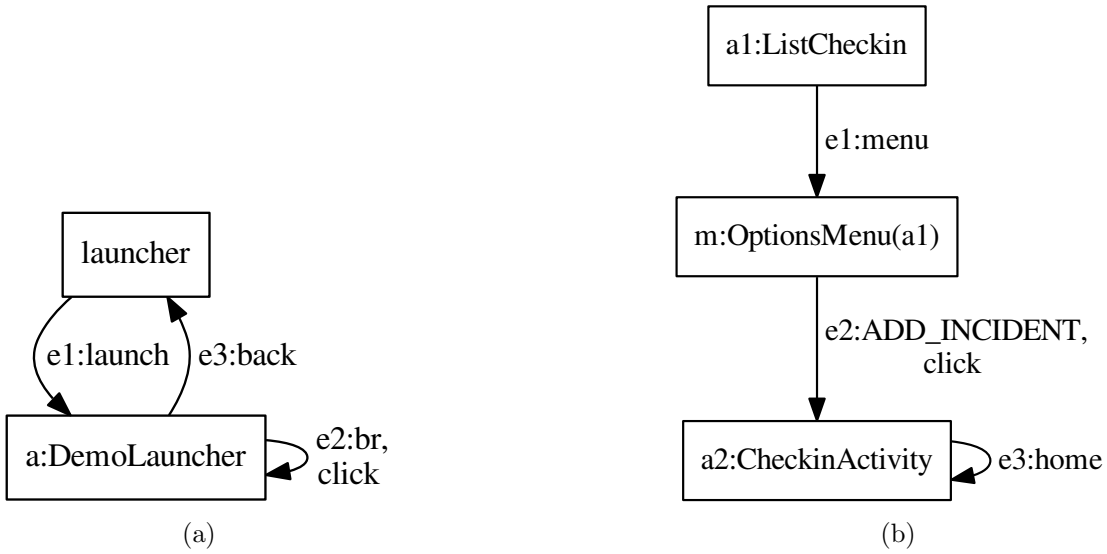

Figure 2.5: WTGs for the running examples.

performed to identify and remove infeasible ICFG edges, based on the knowledge that the calling context of c is o. This analysis is defined in prior work [92], where it was shown to produce more precise control-flow models. For the example in Figure 2.4, this analysis will determine that when `onOptionsItemSelected` is invoked on the item with id `ADD_INCIDENT`, only one branch of the switch statement is feasible and the rest of the branches can be ignored. In addition, we remove ICFG edges related to null pointer checks and throwing of unchecked exceptions, since in our experience they represent unusual control flow that does not contribute to defect detection.

Step 2 An ICFG traversal is performed starting from the entry node of c. This traversal follows interprocedurally-valid paths. During the traversal, whenever an add-listener API call site a is encountered, the points-to set of the listener parameter is used to construct and remember pairs $[a,l]$. (Points-to sets are derived as

described elsewhere [74].) Similarly, we also record all reached pairs [r,l] where r is a remove-listener API invocation statement. For the example in Figure 2.2, analysis of `onClick` and its callees will identify [`requestLocationUpdates`,`EventManager`] where the second element of the pair denotes the listener created at line 12. For Figure 2.4, analysis of `onCreate` will identify a similar pair with the activity being the listener. In addition, analysis of `onDestroy` will detect [`removeUpdates`,`CheckinActivity`].

Step 3 For each [a,l] encountered in Step 2, we need to determine whether the ICFG contains a path from statement a to the exit of c along which there does not exist a matching remove-listener operation [r,l]. If the ICFG contains at least one [r,l] for the same listener l (as determined by Step 2), we perform an additional traversal on the reverse ICFG, starting from the exit node of c. This traversal considers only valid paths with proper matching of call sites and return sites. During the traversal, whenever a remove-listener API call site r with listener l is encountered and it matches the pair [a,l] being considered, the traversal stops. If the add-listener call site a is never reached, this means that [a,l] is not downward exposed and is not included in set $A(c, o)$. In both of our examples, the add-listener operation is downward exposed and this step does not modify sets $A(c, o)$.

Step 4 We construct a similar set $R(c, o)$ of remove-listener operations. However, only operations that are guaranteed to execute along all possible execution paths should be included in this set. If [r,l] could be avoided along some path, this could lead to a leak of listener l. Thus, for each [r,l] observed in Step 2, we perform a traversal of valid ICFG paths, starting from the entry node of c, and stopping if [r,l] is encountered. If the exit node of c is reached, this means that some valid ICFG

path can avoid [r,l]. Set $R(c, o)$ excludes such remove-listener operations. In the running example, the analysis of `onDestroy` will determine that each path through the callback must reach the remove-listener call site, and therefore this operation is included in $R(c, o)$.

▷ *Example:* The final sets A and R computed by Phase 1 for the two running examples are as follows:

$A(\texttt{onResume}, \texttt{DemoLauncher}) = \emptyset$
$R(\texttt{onResume}, \texttt{DemoLauncher}) = \emptyset$
$A(\texttt{onClick}, \texttt{br}) =$
{ [`requestLocationUpdates`,`EventManager`] }
$R(\texttt{onClick}, \texttt{br}) = \emptyset$
$A(\texttt{onOptionsItemSelected}, \texttt{ai}) = \emptyset$
$R(\texttt{onOptionsItemSelected}, \texttt{ai}) = \emptyset$
$A(\texttt{onCreate}, \texttt{CheckinActivity}) =$
{ [`requestLocationUpdates`,`CheckinActivity`] }
$R(\texttt{onCreate}, \texttt{CheckinActivity}) = \emptyset$
$A(\texttt{onDestroy}, \texttt{CheckinActivity}) = \emptyset$
$R(\texttt{onDestroy}, \texttt{CheckinActivity}) =$
{ [`removeUpdates`,`CheckinActivity`] }
$A(\texttt{onStart/onResume/onPause}, \texttt{CheckinActivity}) = \emptyset$
$R(\texttt{onStart/onResume/onPause}, \texttt{CheckinActivity}) = \emptyset$

Here `br` and `ai` represent the static abstractions of the corresponding run-time widgets. ◁

2.3.2 Phase 2: Path Generation

The second phase of the analysis creates a set of candidate paths that represents the lifetime of an activity (for Pattern 1) or the transition to a long-wait state (for Pattern 2). For each activity $w \in \mathbf{Win}$, we consider all incoming WTG edges $t_1 =$ [w',w] that have *push* w as the last element in $\delta(t_1)$. Starting from each such t_1, we perform a depth-first traversal to construct "candidate" paths $\langle t_1, t_2, \ldots, t_n \rangle$. The

Algorithm 1: GenerateCandidatePaths

```
1  foreach activity w ∈ Win do
2      foreach edge t1 = [w′,w] such that δ(t1) ends with push w do
3          path ← ⟨t1⟩
4          stack ← ⟨w⟩
5          Traverse(w, path, stack)
6  procedure Traverse(w, path, stack)
7      if path.length > k then
8          return
9      if ActivityLifetime(path, stack) then
10         record path
11         return
12     if LongWait(path, stack) then
13         record path
14         return
15     foreach edge t = [w,w′] such that t ∉ path do
16         if CanAppend(t, path, stack) then
17             DoAppend(t, path, stack)
18             Traverse(w′, path, stack)
19             UndoAppend(t, path, stack)
```

details of this traversal are presented in Algorithm 1. During the traversal, *path* stores the current path and *stack* is the window stack corresponding to that path. We only consider paths whose length does not exceed some analysis parameter k (in our implementation, this parameter's value is 5). Any path that represents a lifetime for the initial activity w or a transition to a long-wait state from w is recorded for later processing.

Helper function ActivityLifetime checks the following conditions: (1) $\delta(t_n)$ of the last edge t_n in *path* contains *pop* w, and (2) the stack operations in $\delta(t_n)$, up to and including this *pop* w, when applied to *stack*, result in an empty stack. The second condition guarantees that the sequence of push/pop operations from *push* w in $\delta(t_1)$ to *pop* w in $\delta(t_n)$ is a string in the language defined by *Balanced*.

Helper function LongWait determines if *path* will transit to a long-wait state from the initial activity w. The following conditions are checked: (1) *stack* is not empty and its top window w' is either w or a dialog/menu owned by w, and (2) the event on the last edge in *path* is $[w',home]$. Since the window stack is not empty, the sequence of push/pop operations along *path* is a string in the language defined by *Valid*.

During the depth-first traversal, helper function CanAppend (invoked at line 16) considers the sequence $\delta(t)$ of stack operations for a given edge $t = [w,w']$ and decides whether this sequence can be successfully applied to the current window stack. In particular, for each *pop* w'' operation in $\delta(t)$, the current top of the stack must match w''. Furthermore, after all operations are applied, the top of the stack must be the same as the target node of t. If CanAppend returns true, it means that the sequence of stack push/pop operations in the concatenation of $\delta(path)$ and $\delta(t)$ is a string in the language defined by *Valid*.

If transition t is a valid extension of the current path, helper function DoAppend appends t to *path* and applies stack operations $\delta(t)$ to *stack*. After the traversal of the new path completes, helper function UndoAppend removes t from the path and "unrolls" the changes made to *stack* due to operations $\delta(t)$.

▷ *Example:* Consider the example in Figure 2.2 and its WTG shown in Figure 2.5a. Let a denote the WTG node for `DemoLauncher`. Figure 2.5a shows transitions $t_1 = [launcher,a]$, $t_2 = [a,a]$, and $t_3 = [a,launcher]$ with events $\epsilon(t_1) = launch$, $\epsilon(t_2) = [br,click]$ and $\epsilon(t_3) = [a,back]$. In addition, consider transition $t_4 = [a,a]$ with $\epsilon(t_4) = [a,home]$ (not shown in the figure). The stack operations for these four edges are $\delta(t_1) = push\ a$, $\delta(t_2) = [\ ]$, $\delta(t_3) = pop\ a$, and $\delta(t_4) = [\ ]$.

For the sake of the example, suppose we execute Algorithm 1 with $k = 3$. The candidate paths for Pattern 1 are $\langle t_1, t_3 \rangle$, $\langle t_1, t_2, t_3 \rangle$, and $\langle t_1, t_4, t_3 \rangle$. The second path corresponds to the problematic leaking behavior, as discussed earlier. The candidate paths for Pattern 2 are $\langle t_1, t_4 \rangle$ and $\langle t_1, t_2, t_4 \rangle$. For the second path, the callback sequence before the application goes in the background is [`onResume`,a][`onClick`,br] (because no other lifecycle callbacks are defined in the application), and therefore this is also a leaking path. The next phase of the analysis considers all these candidate paths and identifies the ones with leaking callback sequences. ◁

2.3.3 Phase 3: Detection of Leaking Callback Sequences

In this phase, we perform leak detection on candidate paths recorded in Phase 2. First, the relevant callback sequence is extracted from each candidate path. Consider a transition sequence $T = \langle t_1, \ldots, t_n \rangle$ which represents a Pattern 1 candidate path. The relevant callback subsequence of $\delta(T)$ is $[c_1,w] \ldots [c_m,w]$ where c_1 is the creation callback for w and c_m is the termination callback for w (w is the target window of edge t_1). Similarly, for a sequence $T = \langle t_1, \ldots, t_n \rangle$ which is a Pattern 2 candidate path, the relevant subsequence is $[c_1,w] \ldots [c_m,w']$ where c_1 is the creation callback for w and c_m is the last callback before the application goes in the background.

Given a sequence of callbacks $s = [c_1,o_1][c_2,o_2] \ldots [c_m,o_m]$, we consider its sequence $A_1, A_2, \ldots, A_m$ of add-listener sets and $R_1, R_2, \ldots, R_m$ of remove-listener sets. Recall from Definition 1 that s leaks listener l if there exists an add-listener operation $[a,l] \in A_i$ such that for each $j > i$ there does not exist a matching $[r,l] \in R_j$. In Phase 1, we have already computed sets $A(c, o)$ and $R(c, o)$ for any relevant c and o. To detect leaks, we examine each element $[c_i,o_i]$ of s in order and maintain a set L of added but

not yet released listeners. Initially, L is empty. When $[c_i,o_i]$ is processed, all elements of $R(c_i, o_i)$ are removed from L, and then all elements of $A(c_i, o_i)$ are added to L. Any $[a,l]$ that remains in L at the end of this process is considered to be a leak.

▷ *Example:* Consider again Figure 2.2 and the WTG in Figure 2.5a. We have $t_1 = [launcher,a]$, $t_2 = [a,a]$ for the button click, $t_3 = [a,launcher]$ for *back*, and $t_4 = [a,a]$ for *home*. Candidate paths for Pattern 1 (for $k=3$) are $\langle t_1, t_3\rangle$, $\langle t_1, t_2, t_3\rangle$, and $\langle t_1, t_4, t_3\rangle$. For the first and the third path, the relevant callback sequence is [`onResume`,a] which is not leaking because both $A($`onResume`$, a)$ and $R($`onResume`$, a)$ are empty. For the second path, callback sequence [`onResume`,a][`onClick`,br] leaks [`requestLocationUpdates`,`EventManager`]. For candidate path $\langle t_1, t_2, t_4\rangle$ for Pattern 2, the callbacks before the application goes in the background are also [`onResume`,a] [`onClick`,br] and there is a leak as well.◁

Defect reporting For any leaking candidate path $\langle t_1, \ldots\rangle$, the analysis records the pair $[w,l]$ of the initial activity w (i.e., the target of t_1) and the leaking listener l, identified by the allocation site of the corresponding object. For Figure 2.2, this would be [`DemoLauncher`,l_{12}] where l_{12} is the `EventManager` allocation site at line 12 in the code. For each recorded pair, the leaking candidate paths for that pair are also recorded. Each $[w,l]$ is reported as a separate defect, since it requires the programmer to examine the callbacks associated with w and to determine whether they manage listener l correctly.

Defect prioritization In addition to these reports, we also classify leaking listeners as "high" or low "low" priority, based on the following rationale. Consider again the example in Figure 2.4. The leaking behavior can be observed only when a location

read is not obtained (e.g., the weather does not allow a GPS fix), which arguably is not a very frequently-occurring situation. If we analyze `onLocationChanged`—the callback that is executed on a listener l when a location read is obtained—we can determine whether it contains a remove-listener operation for l along each execution path. If this is the case, a location read will release the listener. In the defect report from the analysis such listeners are labeled as "low priority": they should still be examined by the programmer, but perhaps after other leaking listeners have been examined. To make this distinction, for each leaking l we analyze the corresponding callback (`onLocationChanged` or similar method) using the same approach as in Step 4 of Phase 1. The defect in Figure 2.4 will be reported as low priority, while the one from Figure 2.2 will be high priority. In our experiments 3 out of the 17 reported defects were classified as low priority ones.

2.4 Evaluation

The static analysis was implemented in GATOR [24], our open-source static analysis toolkit for Android. The toolkit contains implementations of GUI structure analysis [74, 89] and WTG generation [91–94]. The implementation of the energy defect analysis is currently available as part of the latest release of GATOR.

The goal of our evaluation is to answer several questions. First, how well does the analysis discover GUI-related energy-drain defects already known from prior work? Second, does the analysis discover defects that have not been identified in prior work? Third, does the detection exhibit a reasonably small number of false positives? Finally, what is the cost of the analysis?

Application	WTG			Defects				Time
	Nodes	Edges	Paths	Pat-1	Real-1	Pat-2	Real-2	(s)
`droidar`	10	120	82292	2	2	2	2	2.47
`osmdroid`	14	92	1425	0	0	2	2	0.07
`recycle`	7	22	98	1	1	1	1	0.04
`sofia`	11	55	237	1	1	1	1	0.05
`ushahidi`	42	296	31416	1	1	3	3	1.21
`droidar-f`	10	120	82292	1	1	1	1	2.44
`osmdroid-f`	14	92	1425	0	0	0	0	0.07
`recycle-f`	8	29	258	0	0	0	0	0.04
`sofia-f`	15	67	406	0	0	0	0	0.08
`ushahidi-f`	42	284	30758	0	0	2	2	0.71
`heregps`	3	14	414	1	1	1	1	0.05
`locdemo`	5	13	228	1	1	1	1	0.04
`speedometer`	2	5	10	1	1	1	1	0.03
`whereami`	5	17	51	1	1	1	1	0.03
`wigle`	18	64	3769	1	0	1	1	0.41

Table 2.1: Analyzed applications and detected defects.

Benchmarks To answer these questions, we used several sources of benchmarks, as listed in Table 2.1. First, we considered the benchmarks from the work on GreenDroid [55] that exhibit defects due to incorrect control flow and listener operations in the UI thread of the application. Almost all such GUI defects involve operations related to location awareness, and our static analysis was built to track add/release operations for location listeners. We also considered the fixed versions of these benchmarks—that is, the versions that involve fixes of these known defects. Both defective and fixed versions were obtained from the public GreenDroid web site.[3] In

[3] `sccpu2.cse.ust.hk/greendroid`

Table 2.1, applications in the first part of the table are the defective ones, while applications in the second part of the table, suffixed with `-f`, are the ones with defect fixes.

We also considered the F-Droid repository of open-source applications[4] and searched for applications that use location-awareness capabilities in their UI-processing code. Specifically, the textual description and the manifest file were checked for references to location awareness of GPS, and the code was examined to ensure that the UI thread uses location-related APIs. For the applications we could successfully build and run on an actual device, the static analysis was applied to detect potential defects. Out of the 10 applications that were analyzed, 5 were reported by the analysis to contain defects. The last part of Table 2.1 shows these 5 applications.

Columns "Nodes" and "Edges" show the numbers of WTG nodes and edges, respectively. Column "Paths" contains the number of candidate paths that were recorded and then analyzed for leaking listeners. The last column in the table shows the cost of the analysis; for this collection of experiments, this cost is very low.

Detected defects Recall that for a leaking path $\langle t_1, \ldots \rangle$, the analysis reports a pair $[w,l]$ of the initial activity w (i.e., the target of t_1) and the leaking listener l. We consider each $[w,l]$ to be a defect. Column "Pat-1" shows the number of such defects that were reported by the static analysis as instances of Pattern 1. Column "Pat-2" shows a similar measurement for Pattern 2. In our experiments, a total of 17 unique pairs $[w,l]$ were reported, and all defects that match Pattern 1 (11 defects) also match Pattern 2 (17 defects), but not vice versa. However, it is still useful to detect both patterns statically, as they correspond to two different scenarios. If a defect

[4] `f-droid.org`

matches both Pattern 1 and Pattern 2 (e.g., the one in Figure 2.2), it usually means that the programmer completely ignored the removal of the listener. On the other hand, if a defect matches Pattern 2 but not Pattern 1 (e.g., the one in Figure 2.4), this means that the programmer attempted to remove the listener, but did not do it correctly. Given the low cost of the analysis, we believe that detection of both patterns is valuable.

Column "Real-1" shows the number of detected defects from column "Pat-1" that we manually confirmed to be real, by observing the actual run-time behavior of the application. Similarly, column "Real-2" shows the number of defects from column "Pat-2" that were verified in the same manner. Only one reported defect is incorrect: in `wigle`, a defect is incorrectly reported by the analysis as an instance of both Pattern 1 and Pattern 2, while in reality it is only an instance of Pattern 2. The cause of this imprecision will be discussed shortly.

Two conclusions can be drawn from these measurements. First, the analysis successfully detects various defects across the analyzed applications. Even the "fixed" versions are not free of defects: for example, we discovered two defects in `ushahidi-f` that were not reported in the work on GreenDroid, and were missed by the application developers when `ushahidi` was fixed to obtain `ushahidi-f` (in fact, these two defects are quite similar to the one that was fixed). A similar situation was observed for `droidar`. This observation indicates the benefits of static detection, compared to run-time detection which depends on hard-to-automate triggering of the problematic behavior. Of course, static detection has it own limitations, with the main one being false positives. However, the experimental results for the 15 benchmarks shown in Table 2.1 indicate that the proposed analysis achieves very high precision.

Application	$\|D\|$	$\|S\|$	$\|D-S\|$	$\|S-D\|$
`droidar`	1	2	0	1
`osmdroid`	2	2	0	0
`recycle`	1	1	0	0
`sofia`	1	1	0	0
`ushahidi`	1	3	0	2

Table 2.2: Defects reported: GreenDroid (D) vs static analysis (S).

False positive The false positive for `wigle` is because the developer decided to override standard method `Activity.finish` with a custom version which removes the listener. When method `finish` is invoked on an activity (by the application code or by the framework code), this causes the termination of the activity. However, this method is not a callback that is defined as part of the lifecycle of an activity, and is rarely overridden by applications. In other words, `finish` can be called to force termination, but it is not executed as part of the actual termination process. Thus, `finish` does not appear on WTG edges (although it is accounted for during WTG creation [93, 94]). In fact, termination could happen even if `finish` is not called: for example, the system may silently terminate an activity to recover memory [26]. The Android lifecycle model guarantees that `onDestroy` will be called in all scenarios, and this is where the listener should be removed, rather than in `finish`. This example indicates that the developer misunderstands the activity lifecycle. During the manual examination of this defect on a real device we did observe that the location listener is properly released, and decided to classify the defect as a false positive, although one could argue that it violates Android guidelines.

Comparison with GreenDroid To compare the proposed static detection with the most relevant prior work, Table 2.2 considers the UI thread defects [w,l] reported by the dynamic analysis approach in GreenDroid. For a given application, let D be the set of these defects, while S be the set of defects reported by our static analysis. The sizes of these sets are given in the second and third column in Table 2.2. The next two columns show the sizes of sets D–S and S–D, respectively. As the next-to-last column shows, our static analysis reported *all* defects from the prior dynamic analysis work. The last column shows how many of the statically-detected defects were *not* reported by GreenDroid; for two of the applications, there are additional defects we discovered statically (and these defects are still present in the fixed versions from GreenDroid). A possible explanation of this result is that the run-time exploration strategy in GreenDroid did not trigger the necessary GUI events; in general, comprehensive run-time GUI coverage is challenging [18].

Overall, these results indicate that static detection could be more effective than dynamic detection. At the same time, it is important to consider the relative strengths and weaknesses of both approaches: while the static analysis can model more comprehensively certain behaviors of the UI thread, other aspects of run-time semantics are not modeled statically (e.g., asynchronous tasks and services) and dynamic analysis does capture additional defects for such behaviors. This highlights the need for more comprehensive static control-flow analyses for Android, as well as hybrid static/dynamic approaches for defect detection.

2.5 Conclusions

We propose a static analysis for detection of energy-related defects in the UI logic of an Android application. The technical foundation for this analysis is the static modeling of possible sequences of window transitions and their related callbacks. By identifying certain such sequences, based on the state of the window stack, we define two patterns of behavior in which location listeners are leaking. Control-flow analysis of individual callbacks is combined with analysis of callback sequences to identify instances of these patterns. Seventeen known and new defects were detected in previously-analyzed and newer-analyzed applications. All but one of the reported defects are observable at run time. The evaluation also shows that the cost of the analysis is low.

CHAPTER 3: Static Detection of Sensor-Related Energy Defects

Android devices have hardware sensors for acceleration, rotation, proximity, light, etc. As a general guideline, the app should disable sensors that are not needed. Failing to disable sensors—that is, sensor leaks—can drain the device battery.

We propose a static analysis to detect potential sensor leaks in Android apps. The leaks are then verified by generated test cases. The approach was implemented in the SENTINEL tool for sensor testing to detect leaks. The tool and benchmarks used for its evaluation are publicly available at https://presto-osu.github.io/Sentinel. Section 3.2.1 describes our static graph model of sensor-related objects and API calls. Graph edges are labeled with symbols representing the opening/closing of UI windows and the acquiring/releasing of sensors. Section 3.2.2 defines a *context-free-language reachability* (CFL-R) problem over the graph. This problem is based on two context-free languages over the alphabet of symbols. A graph path that defines a string from these languages is a "witness" of a leak. Given this CFL-R formulation, Section 3.2.3 describes an approach to identify and report buggy paths. This approach traverses selected CFL-R paths and checks them for leaks. The reported paths are then used to generate test cases, as outlined in Section 3.2.4. Our experimental results, presented in Section 3.4, indicate that the proposed approach achieves high precision and exhibits practical cost.

3.1 Android GUIs and Sensors

3.1.1 Android GUI Control Flow

Details of Android GUI structure and behavior were already discussed in Section 2.1. This section illustrates the relevant control flow abstractions with the following example.

▷ *Example:* Figure 3.1 shows a simplified example derived from a sensor leak we found in the apps we analyzed. Calculator Vault is a vault app used to hide photos and other documents. The app has over a million downloads in the Google Play Store. The example shows two of the app's activities: `SettingActivity` and `UnlockActivity`. The first activity has a button widget (`btn` at line 4); the second one has a switch widget (`sc` at line 17) which is a toggle to select between two options. if `btn`'s button is touched, `onClick` (lines 7–13) is invoked by the Android platform code. In this example, using `startActivity` at line 12, the event handler opens a new window corresponding to `UnlockActivity`. Inside `UnlockActivity`, when the user changes the state of switch `sc`, `onCheckedChanged` (lines 24–27) is invoked. As discussed later, this event handler registers a listener for the accelerometer sensor. ◁

3.1.2 Sensors in Android Apps

On an Android device there exists multiple categories of sensors. Each category is represented by an integer constant defined in class `android.hardware.Sensor`. For example, `Sensor.TYPE_ACCELEROMETER` corresponds to all accelerometer sensors. A sensor object, instantiated from `android.hardware.Sensor` is used to represent a hardware sensor. These sensor objects are created by the Android framework and will not be replaced or destroyed unless the app process is killed. From our case

```
1  class SettingActivity
2      extends Activity implements OnClickListener {
3    onCreate(...) {
4      Button btn = findViewbyId(R.id.rl_unlockSetting);
5      btn.setOnClickListener(this); ...
6    }
7    onClick(View v) {
8      switch(v.getId()) {
9        ...
10       case R.id.rl_unlockSetting:
11         Intent i = new Intent(UnlockActivity.class);
12         startActivity(i); break;}
13   }
14 }
15 class UnlockActivity extends Activity {
16   onCreate(...) {
17     SwitchCompat sc = ...;
18     SensorManager sm = ...;
19     Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
20     SensorEventListener shakeListener = new SensorEventListener {
21       onSensorChanged(...) {
22         if (...) { sm.unregisterListener(this); }}};
23     sc.setOnCheckedChangeListener(new OnCheckedChangeListener{
24       public onCheckedChanged(View v) {
25         ...
26         sm.registerListener(shakeListener, accel);}});
27   }
28   onDestroy() { ... }
29 }
```

Figure 3.1: Example derived from Calculator Vault.

studies, we observed that developers rarely use more than one sensor from a sensor category: typically, only the default sensor is used. To obtain it, the code calls method `getDefaultSensor` defined in `android.hardware.SensorManager`. In Figure 3.1, line 19 illustrates such a call: `accel` refers to the default accelerometer sensor object,

which is used by the application to detect when the user shakes the device in order to unlock it.

To obtain sensor data, the programmer creates and registers a *sensor event listener*. Such a listener is an instance of `android.hardware.SensorEventListener` (line 20). Callback method `onSensorChanged` is invoked on this listener whenever new sensor data is available. Line 26 shows how a listener is registered with a sensor object. The sensor hardware will be enabled when there exists any listener registered to listen to the sensor's changes. The hardware will be turned off when all listeners are removed via `unregisterListener` (illustrated at line 22).

The sensor leak in the running example occurs as follows. After `UnlockActivity` is opened, the user may toggle `sc`'s switch in the UI, which will invoke `onCheckedChanged` and as a result will (1) register `shakeListener`'s listener object with `accel`'s sensor object, and (2) wait for a shake gesture from the user to unlock the vault. Whenever the device is moved, `onSensorChanged` is invoked with information about the physical movement. If this movement is above some threshold (checked at line 22), it is considered to be "shake to unlock" which releases the listener via `unregisterListener` and unlocks the vault. However, if the user does not shake the device, the listener will continue to listen for updates. If the user quits this app and makes the phone stationary, `UnlockActivity` will be closed. At that time, lifecycle callback `onDestroy` (line 28) does not release the sensor either. Thus, the window that acquired the sensor does not release it, which keeps the sensor alive and drains the battery. We have confirmed this behavior using tests on a real Android device. This is an example of a typical sensor leak pattern. Our goal is to express such sensor leaks formally and to detect them statically.

3.2 Control-Flow Analysis for Sensor Leaks

3.2.1 Control-Flow Model

Window transition graph The run-time behavior of windows and events can be modeled statically in a variety of ways. The starting point of our work is the *window transition graph* (WTG) [93, 94] static model discussed earlier. Recall that graph nodes graph represent windows and an edge $e = w_i \rightarrow w_j$ indicates that when window w_i was visible and interacting with the user, some user-triggered event caused window w_j to be displayed and to begin interacting with the user. It is possible that $i = j$, in which case the currently-active window does not change.[5] As discussed earlier, during a transition from w_i to w_j, various callback methods are executed. The WTG defined in prior work [93, 94] contains information about both callback methods and window open/close effects.

Sensor effects control-flow graph For the purposes of control-flow analysis for sensor leak detection, we define a static model derived from (1) the WTG and (2) further analysis of the callback methods along WTG edges. We refer to this model as the *sensor effects control-flow graph* (SG). The graph is $SG = (N, E, L)$ where N and E are the node set and edge set from the WTG and $L : E \rightarrow \Sigma^*$ defines a label $l(e)$ for each edge $e \in E$. The label is a sequence of symbols from the alphabet

$$\Sigma = \{\mathsf{open}(w_i), \mathsf{close}(w_i), \mathsf{acquire}(s_k), \mathsf{release}(s_k)\}$$

Here symbols $\mathsf{open}(w_i)$ and $\mathsf{close}(w_i)$ denote the opening/closing of a window represented by $w_i \in N$. Symbols $\mathsf{acquire}(s_k)$ and $\mathsf{release}(s_k)$ denote the acquiring

[5] In some cases (e.g., when the device is rotated) the current window is destroyed and then recreated with a different layout. Such cases are also represented as $w_i \rightarrow w_i$ transitions.

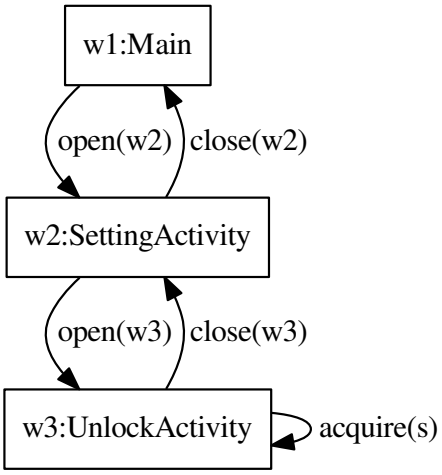

Figure 3.2: SG graph for the running example.

and release of a sensor s_k. The set of sensor abstractions s_k will be described later in this section. To obtain all acquire/release symbols in Σ, and to determine how such symbols appear in label $l(e)$ for an edge e, we analyze the bodies of the callback methods executed during the transition represented by e. This analysis is described in Section 3.2.5. Note that some WTG edges may not have any effects that correspond to the symbols in Σ (that is, $l(e) = \epsilon$); in such cases, the edges are not included in SG.

Example Figure 3.2 shows SG for the running example. Node w_1 corresponds to activity `Main`, which is not shown in the code from Figure 3.1. A widget event handler in w_1 opens `SettingActivity`. The self-edge for w_3 corresponds to a change in the state of switch widget `sc`; the invoked callback `onCheckedChanged` acquires the default accelerometer sensor, denoted by s in the figure. Note that this example is rather simple. However, we have seen many apps where a single edge contains several

symbols (e.g., it represents the opening/closing of several windows, or the acquiring of several sensors).

3.2.2 CFL-Reachability for Sensor Leaks

Given a directed graph with labeled edges, *context-free language (CFL) reachability* defines a set of paths in the graph. The paths are based on a context-free language L over the alphabet Σ of edge labels. An L-path is such that the sequence of edge labels along the path forms a string in L. Many static analyses can be formulated as CFL-reachability problems [73, 78].

Since traditional CFL reachability assumes graph edges labeled with elements of Σ rather than with elements of Σ^+, for the sake of presentation we assume that SG is modified to ensure that each $l(e)$ contains at most one symbol. This can be easily achieved by introducing intermediate graph nodes/edges to "break up" each sequence $l(e)$. Note that this transformation is just a conceptual vehicle for ease of explanation. Our analyses are implemented to handle the general case of $l(e) \in \Sigma^+$.

Using CFL reachability, next we define two sensor leak patterns. These patterns are similar to GPS leaks that have been observed in prior work [85]. However, that work did not consider sensors and did not formulate leak properties and analysis algorithms using CFL reachability.

Leaks Beyond Window Lifetime

Given SG, we define sensor leak analyses as CFL-reachability problems over SG. We start by defining a context-free language $L_1(w_i)$ describing paths that represent the lifetime of a window w_i. By intersecting this language with several regular languages over sensor acquire/release effects, we will capture one common pattern of

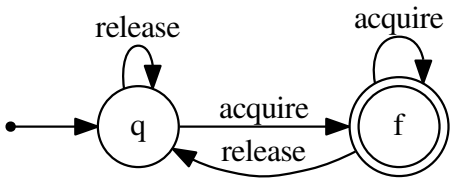

Figure 3.3: Finite automaton $\mathcal{F}$.

sensor leaks. The subscript indicates that this is the first pattern being considered. A second pattern, described later, will be based on another context-free language $L_2(w_i)$.

$L_1(w_i)$ is similar to classic balanced-parentheses languages:

$$\begin{array}{rcl} S_1 & \rightarrow & \mathsf{open}(w_i)\ Bal\ \mathsf{close}(w_i) \\ Bal & \rightarrow & \mathsf{open}(w_j)\ Bal\ \mathsf{close}(w_j) \mid Bal\ Bal \mid Sen \mid \epsilon \\ Sen & \rightarrow & \mathsf{acquire}(s_k) \mid \mathsf{release}(s_k) \end{array}$$

A string in this language corresponds to a run-time execution scenario in which w_i is opened, a number of other windows are opened and closed, and at the end w_i itself is closed. The actual run-time execution is based on a stack of currently-active windows, with the top of the stack being the window that is currently visible and interacting with the user. During the execution described by an $L_1(w_i)$ string, w_i is pushed on top of the stack, additional push/pop operations are performed on top of w_i, and at the end w_i is popped from the stack. Any string from the language describes a possible lifetime for w_i.

To define the correct behavior for sensor effects, we define a regular language $R(s_k)$ for each sensor s_k. We specify this language using a deterministic finite automaton $\mathcal{F} = (Q, \Sigma, \delta, q, f)$. Here $Q = \{q, f\}$ is the set of states, with q being the initial state and f being the final state. The input alphabet is the set Σ defined earlier. The transition function $\delta : Q \times \Sigma \rightarrow Q$ is shown in Figure 3.3. The figure shows

only transitions for symbols acquire and release for the sensor of interest s_k. For the remaining symbols in Σ (open/close, as well as acquire/release for other sensors), there are self-transitions in both states.

If a string belongs to the regular language defined by $\mathcal{F}$, it represents a leak of sensor s_k. Note that in Android it is possible to perform successive acquire operations on the same sensor without in-between release operations; the second, third, etc. acquire have no effect. Similarly, it is possible to have successive release operations without in-between acquire; all but the first release are no-ops. Finally, it is also possible to execute release operations on a sensor that was never acquired. All these scenarios are captured by $\mathcal{F}$.

Since the intersection of a context-free language with a regular language is a context-free language, the language

$$P_1(w_i, s_k) = L_1(w_i) \cap R(s_k)$$

is context-free and thus suitable for CFL-reachability analyses. If there exists an SG path whose edge labels form a string from $P_1(w_i, s_k)$, the lifetime of window w_i acquires sensor s_k without releasing it, and thus matches our first pattern of sensor leaks.

Leaks in Suspended State

The second pattern of sensor leaks will be illustrated using the example in Figure 3.4. CSipSimple is an open-source VoIP app that has been used by several commercial VoIP app which have more than a million downloads on Google Play Store. `InCallActivity` will register a listener for the proximity sensor in `onCreate` and will release this listener in `onDestroy`. This example does not exhibit the leak pattern

```
1  class InCallActivity extends Activity {
2    CallProximityManager proximityManager = ...;
3    onCreate(...) { proximityManager.startTracking(); ... }
4    onResume() { ... }
5    onPause() { ... }
6    onDestroy() { proximityManager.stopTracking(); ... }
7  }
8  class CallProximityManager implements SensorEventListener {
9    SensorManager sm = ...;
10   Sensor proximitySensor = sm.getDefaultSensor(Sensor.SENSOR_PROXIMITY);
11   onSensorChanged(...) { ... }
12   startTracking() {
13     sm.registerListener(this, proximitySensor); ... }
14   stopTracking() {
15     sm.unregisterListener(this); ... }
16 }
```

Figure 3.4: Example derived from CSipSimple.

described earlier: by the time the activity is destroyed, the sensor is released. However, another possible scenario is when the activity is suspended for a long period of time (e.g., hours). For example, if the user presses the HOME button, the app is put in the background but the sensor is still active.

To formalize this second pattern of sensor leaks, we add to the alphabet Σ a symbol $\mathsf{suspend}(w_i)$ for each window w_i. Graph SC is augmented as follows: for each window w_i a new node $\bar{w}_i$ is added to represent the suspended state of w_i. An edge $w_i \rightarrow \bar{w}_i$ is labeled with symbols from Σ representing the sensor effects of lifecycle callbacks (e.g., `onPause`) executed before entering the suspended state. The last symbol on the edge is $\mathsf{suspend}(w_i)$. Another edge $\bar{w}_i \rightarrow w_i$ captures the sensor effects of resuming the app (e.g., sensors being reacquired in lifecycle callback `onResume`). Figure 3.4

illustrates these two callbacks at lines 4 and 5. In this app, both callbacks have no effect on sensors.

As before, we define a context-free language to express how a window w_i reaches a suspended state. This language $L_2(w_i)$ is:

$$\begin{array}{rcl} S_2 & \rightarrow & \mathsf{open}(w_i)\ \mathit{Val}\ \mathsf{suspend}(w_k) \\ \mathit{Val} & \rightarrow & \mathsf{open}(w_j)\ \mathit{Val} \mid \mathit{Bal}\ \mathit{Val} \mid \epsilon \end{array}$$

where Bal was defined earlier. Here Val represents a valid sequence of symbols which could have not-yet-matched open symbols. Language $P_2(w_i, s_k) = L_2(w_i) \cap R(s_k)$ captures the scenario where execution is suspended without releasing sensor s_k. This is the second sensor leak pattern we consider.

3.2.3 Detection and Reporting of Leaks

For any w_i, language $L_1(w_i)$ can be recognized by a pushdown automaton $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, s, f)$. Here $Q = \{s, q, f\}$ is the set of states, with s being the initial state and f being the final state. The input alphabet is the set Σ defined earlier. The stack alphabet $\Gamma = \{\mathsf{open}(w_j), \mathsf{close}(w_j)\}$ for all possible windows w_j (including the "window of interest" w_i). The transition relation δ is shown in Figure 3.5. Each edge is labeled with $(a, x, y) \in \Sigma \times \Gamma \times \Gamma$, indicating that when the input symbol is a and the top of the stack is x, a is consumed, x is popped, and y is pushed on the stack. If $x = \epsilon$, the transition occurs regardless of what is on the stack. If $y = \epsilon$, nothing is pushed on the stack. In Figure 3.5 w_i is the window of interest w_i and w_j is any window (including w_i). A string is accepted in state f with an empty stack.

Note that this automaton is non-deterministic: from state q with symbol $\mathsf{close}(w_i)$ and stack top $\mathsf{open}(w_i)$, there are two possible transitions (to q and to f). The non-determinism can be easily eliminated by introducing an artificial stack symbol $\perp$ (for

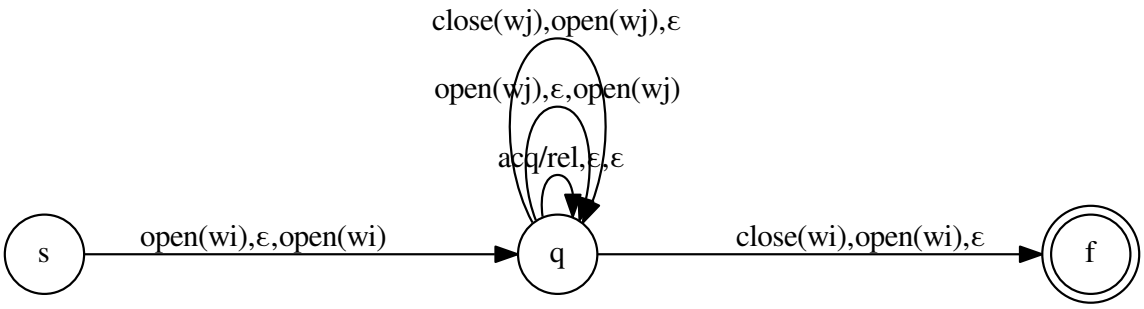

Figure 3.5: Pushdown automaton $\mathcal{P}$.

"bottom of stack") and artificial initial/final states; for brevity, we omit this simple refinement.

The simplest way to recognize language $P_1(w_i, s_k)$ is to run concurrently the pushdown automaton for $L_1(w_i)$ and the (deterministic) finite automaton for $R(s_k)$. Since typically there would be several possible sensors s_k, several finite automata for the corresponding $R(s_k)$ would be maintained. In essence, for each window w_i, this approach traverses each $L_1(w_i)$ path and checks it for leaks of each possible s_i. Since the number of paths is typically infinite, we define a finite subset of paths using two criteria: (1) a path cannot contain the same edge more than once, and (2) the number of open and close symbols along a path cannot exceed a certain pre-defined limit k. The second criterion captures the complexity of sequences of GUI control-flow events, regardless of how these events affect sensors. These two restrictions control the cost of the analysis and the intricacy of "buggy" paths that are eventually reported to a programmer. The recognition of $P_2(w_i, s_k)$ can be done similarly, using a

slightly modified version of the pushdown automaton. In the next section we present additional details on the analysis algorithms.

3.2.4 Generation of Test Cases

For any w_i, all path strings in language $P_1(w_i, s_k)$ can be determined by traversing SG paths starting at w_i and maintaining a stack corresponding to window open/close events. The stack elements are open and close symbols. A new open symbol is pushed on top of the stack. A new close(w_j) is allowed only if the current stack top is open(w_j) for the same window w_j; as a result, the top stack element is popped. During path traversal, the state of finite automaton $R(s_k)$ is updated based on symbols acquire and release. Since typically there would be several possible sensors s_k, several finite automata for the corresponding $R(s_k)$ would be maintained. The path strings for $P_2(w_i, s_k)$ can be generated similarly. Since the number of paths is typically infinite, we define a finite subset of paths using two criteria: (1) a path cannot contain the same edge more than once, and (2) the number of open and close symbols along a path cannot exceed a certain pre-defined limit k (our implementation uses $k = 4$). The second criterion captures the complexity of sequences of GUI control-flow events, regardless of how these events affect sensors. These two restrictions control the number and length of generated test cases.

Given an SG path generated as described above, it can be mapped to a sequence of GUI events using information available in the WTG. For example, for the graph in Figure 3.2, the path with edge labels open(w_3), acquire(s), close(w_3) will be mapped to the test case shown in Figure 3.6. The test case uses a Python wrapper for Google's UI Automator testing framework [81]. Several low-level details of the test case are

```
d.screen.on()  # turn on device screen
killApp("com.calculator.vault") # kill target app to clean up acquired resources
oldsensors = readAssociatedSensors() # gather currently-acquired sensors
startActivity("com.calculator.vault", "com.calculator.vault.UnlockSettingActivity")
d(resourceId="com.calculator.vault:id/shake_btn").click() # click the switch widget
d.press.back() # press the back button
newsensors = readAssociatedSensors() # gather acquired sensors after execution
# report differences between newsensors and oldsensors
```

Figure 3.6: Example of generated test case.

omitted for brevity. Section 3.3 provides additional details on how such test cases are generated and executed.

Test case filtering We employ three filtering techniques to reduce the number of generated test cases. First, note that all paths in a particular set $P_1(w_i, s_k)$ are in some sense equivalent: they exhibit the same pattern, for the same window w_i and sensor s_k. Thus, after leaking paths are generated, we select only one path from each $P_1(w_i, s_k)$ set for test generation—specifically, any minimal-length path in that set. Similar filtering is applied to any $P_2(w_i, s_k)$.

Next, consider SG for the running example. The path with labels $\mathsf{open}(w_3)$, $\mathsf{acquire}(s)$, $\mathsf{close}(w_3)$ is in language $P_1(w_3, s)$. But the path with labels $\mathsf{open}(w_2)$, $\mathsf{open}(w_3)$, $\mathsf{acquire}(s)$, $\mathsf{close}(w_3)$, $\mathsf{close}(w_2)$ is in language $P_1(w_2, s)$. It is redundant to generate test cases for both paths: from the point of the view of a programmer, the "blame" should be assigned to activity w_3 because that activity was responsible for acquiring (but not releasing) the sensor. Thus, only a test case for the first path should be generated and executed.

To achieve this filtering, during the traversal of an $L_1(w_i)$ path we ignore $\mathsf{acquire}(s_k)$ if w_i was not responsible for acquiring s_k. To make this decision, we consider the state

of the open/close stack at the time when acquire(s_k) was encountered. If the top of the stack is not open(w_i), the acquire operation is ignored.[6] This guarantees that any leaking s_k reported due to $P_1(w_i, s_k)$ can be blamed on w_i. The same filtering is used for $P_2(w_i, s_k)$.

The last filter we apply is for $L_2(w_i)$ paths. While the definition of this language allows strings starting with open(w_i) and ending with suspend(w_j), we only report paths for which w_j is either the same as w_i, or a menu/dialog acting on behalf of w_i (menus and dialogs will be described shortly). If w_i is responsible for acquiring s_k and there exists any leaking $L_2(w_i)$ path, there also exists a leaking $L_2(w_i)$ path that satisfies this constraint.

3.2.5 Static Sensor-Related Abstractions

A sensor s_k described earlier is actually a pair $\langle l, o \rangle$ of a sensor listener l and a sensor object o. For example, in Figure 3.1 the sensor being analyzed is a pair of the `SensorEventListener` object l referenced by `shakeListener` and the `Sensor` object o referenced by `accel`. To determine these s_k, our analysis first creates static abstractions of `Sensor` objects. One static object o per sensor type (e.g., accelerometer, proximity) is created. Next, propagation for integer constants `Sensor.TYPE_*` is used to determine which sensor types reach calls to `getDefaultSensor`. The resulting sensor objects o returned by such calls are then propagated to calls to `registerListener`.

The listener objects are created by instantiating classes that implement interface `SensorEventListener`. Each such `new` expression corresponds to a static listener object l. These objects are also propagated to calls to `registerListener`. For every

[6] More generally, the top of the stack could be open(w_j) for some menu or dialog w_j working on behalf of activity w_i. This generalization is discussed in Section 3.3.

l and o that reach some such call, the analysis creates a corresponding sensor abstraction $s_k = \langle l, o \rangle$. Each such call is considered an instance of an "acquire" operation for s_k. Similarly, calls to `unregisterListener` are instances of "release" operations. Note that in both Figure 3.1 and Figure 3.4, the call to `unregisterListener` takes as a parameter the listener but not the sensor object. This method has two versions: one that takes as parameters both l and o, and another that takes only l. In the latter case, the call is considered to be a release operation for any $s_k = \langle l, \ldots \rangle$.

Recall that our SG control-flow model (illustrated in Figure 3.2) is derived from the WTG. Recall that the WTG edges are labeled with information about the callbacks invoked at run time—e.g., the lifecycle callbacks `onCreate`/`onDestroy` and the event handler `onCheckedChanged` in Figure 3.1. Each such callback is analyzed to determine whether it contributes any $\mathsf{acquire}(s_k)$ or $\mathsf{release}(s_k)$ symbols to the corresponding SG edge.

This analysis of a callback method m considers m and its transitive callees in the app code. The callees are determined using class hierarchy analysis [20]. If any one of those methods contains an acquire operation for some s_k, it is necessary to check whether there is an interprocedural path from that operation to the exit of m that is free of a corresponding release of s_k. If such a path exists, callback m contributes symbol $\mathsf{acquire}(s_k)$. Callbacks `onCheckedChanged` in Figure 3.1 and `onCreate` in Figure 3.4 are examples of this case. It is also necessary to check whether every interprocedural path from the entry to the exit of m contains a release operation for s_k. If this is the case, the execution of m is guaranteed to release s_k and the callback contributes symbol $\mathsf{release}(s_k)$. Callback `onDestroy` in Figure 3.4 illustrates this case. Note that a callback m could contribute both a release operation and an

acquire operation for the same s_k (e.g., if it releases the sensor and then re-acquires it). In this case the analysis of m results in the string $\mathsf{release}(s_k), \mathsf{acquire}(s_k)$.

In addition to lifecycle callbacks and GUI event handler callbacks, we also need to consider the potential effects of callback `onSensorChanged`. Lines 21–22 in Figure 3.1 illustrate this callback. Whenever a listener l is registered with a sensor object o, the listener is (almost) immediately notified of the current value of the sensor data, via an invocation of `onSensorChanged` on l. It is possible that this invocation unconditionally releases the sensor, and we have seen such examples in real apps. To account for this possibility, for each $\mathsf{acquire}(s_k)$ where $s_k = \langle l, o \rangle$, we identify the corresponding callback `onSensorChanged` for l and analyze it using the same callback analysis described earlier. The contributions of the callback are appended at the end of each $\mathsf{acquire}(s_k)$ symbol in the control-flow model. For the example in Figure 3.1, the callback contains a release operation but this operation does not occur along every path, due to the `if` statement. If, hypothetically, the callback did *not* contain this `if` statement, it would contribute a symbol $\mathsf{release}(s)$. In this case, the self-edge for w_3 in Figure 3.2 would be labeled with the string $\mathsf{acquire}(s), \mathsf{release}(s)$.

3.3 Analysis Implementation

The analysis described in the previous section was implemented in the Soot analysis framework [77]. The starting point of the implementation is the publicly-available GATOR analysis toolkit for Android [24] which contains an implementation of the window transition graph (WTG) representation [93, 94] described in Section 3.2.1. Rather than explicitly building the sensor effects control-flow graph SG, our implementation works directly on the WTG. A WTG edge $w \rightarrow w'$ shows that from

current window w can transition to window w'. Such an edge contains the sequence of UI event handler callbacks and window lifecycle callbacks. In addition, an edge describes the window open/close effects, from which symbols $\mathsf{open}(w_i)$ and $close(w_i)$ can be directly derived. The analysis works in two stages, as described below.

Stage 1: Sensors and sensor operations To determine how $\mathsf{acquire}(s_k)$ and $\mathsf{release}(s_k)$ symbols should be introduced, the analysis first constructs a set of sensor abstractions s_k. Each $s_k = \langle l, o \rangle$ for a sensor listener l and a sensor object o. As described in Section 3.2.5, l corresponds to a `new` expression for `SensorEventListener` and o corresponds to a sensor type. Integer constants such as `Sensor.TYPE_ACCELEROMETER` are propagated to calls to `getDefaultSensor` to determine which o is being produced. Objects l and o are then propagated to calls to `(un)registerListener`. The propagation of integer constants and object references is done in a flow/context-insensitive manner, based on an internal representation similar to the pointer assignment graph used in Soot [44]. The result is a set of s_k abstractions as well as the program statements that acquire/release them.

Next, analysis of UI event handler callbacks and lifecycle callbacks along WTG edges is performed to determine whether any $\mathsf{acquire}(s_k)$ or $\mathsf{release}(s_k)$ symbols are contributed by each callback method. As outlined in Section 3.2.5, this involves interprocedural reachability to/from acquire and release operations. The reachability computation uses Soot's control-flow-graph representation for each reachable method. Calls are resolved using class hierarchy information. Callbacks `onSensorChanged` and their transitive callees are analyzed in a similar manner.

Stage 2: Generation and checking of SG paths For each activity w_i, the analysis considers the context-free language $L_1(w_i)$ defined in the previous section. (The analysis for $L_2(w_i)$ is done similarly.) Recall that this language describes a possible lifetime for w_i. To reduce the number and complexity of paths being considered, we only consider a path if it does not contain duplicated edges and if its number of open and close symbols does not exceed an analysis parameter k (Section 3.2.3). In addition, as a pre-processing step, we identify equivalence sets of SG edges: if two edges have the same source, target, and label, they are equivalent. Only one edge per equivalence class is considered when exploring paths.

The analysis performs a graph traversal starting from an open(w_i) edge and runs the pushdown automaton $\mathcal{P}$ (Figure 3.5) along the current path. The state of $\mathcal{P}$ is updated when a new edge is added to the path, assuming the edge is acceptable according to $\mathcal{P}$ and the restrictions from the previous paragraph. The path ends with close(w_i) and an empty stack in $\mathcal{P}$. Each generated path from $L_1(w_i)$ is checked with finite automaton $R(s_k)$ for each s_k for a possible leak of s_k.

Stage 3: Test generation and execution Paths generated in Stage 2 are used to generate test cases. Test generation maps an SG path to a sequence of calls to UIAutomator API calls. Widgets are referenced using their ids defined in XML layout files or in `setId` calls in the code, as determined by GATOR [74]. If widgets do not have static ids (e.g., list items), a test case cannot be generated. For widgets that require user input (e.g., `EditText`), manual post-processing is needed; we have seen a very small number of cases in which this occurs.

The generated test cases are executed using a Python wrapper for UI Automator [81, 82]. This framework allows testing to be controlled from a computer with direct access to the Android Debug Bridge (`ADB`), which is necessary for run-time sensor leaks measurements. Given a path reported by the static analysis, SENTINEL generates code which sets up the test case, starts the first activity on the path using an Android intent, and triggers the necessary GUI events. A simplified example of such code was presented in Figure 3.6. Depending on the application, it may be necessary to perform additional steps by the tester to fully set up the test case: for example, for the calculator vault app, it is necessary to setup a password for unlocking before the rest of the app can be used.

Acquired sensors with information about listener's package names and sensor types can be queried using `dumpsys` command in `ADB`. Each generated test case performs this measurement at the start and at the end of its execution (`readAssociatedSensors` in Figure 3.6). If a sensor is not at the start but is active at the end of the test case, and if the listener's package name is the same as the target application, a leak report will be generated. In our experiments, we executed the test cases and observed the sensor on a Google Nexus 5X smartphone with Android 7.1.2.

Handling of menus and dialogs Activities are the primary windows in Android apps. However, the UI also allows for menus and dialogs, which are windows used to provide helper functionality for an activity. For example, in Figure 3.1, the developer could have chosen to add a dialog window that is opened when `btn`'s button is pressed, in order to ask for confirmation that the vault should be locked. Upon user confirmation, the dialog's event handler would have started the sensor event listener.

In general, menus and dialogs have their own widgets, UI event handlers, and lifecycle callbacks. Our implementation handles all these features; for example, these callbacks are analyzed to determine acquire/release symbols as described earlier. Languages P_1 and P_2 are only considered for activities because menus and dialogs are short-lived and a sensor they have acquired on behalf of some activity may be still active past their lifetime.

One adaptation needed is in reporting of leaks. Recall from Section 3.2.3 that blame is assigned to an activity w_i only if it was the currently-active window when a leaking s_k was last acquired. It is possible that a menu or a dialog was the actual active window that acquired s_k on behalf of activity w_i. Typically w_i is the last still-opened activity at the time when the menu/dialog was opened. Instead of recording the menu/dialog for subsequent blame assignment, we examine the stack of $\mathcal{P}$ from top to bottom for the first symbol $\mathsf{open}(w_i)$, where w_i is an activity, and then record w_i. This ensures that only activities are eventually reported by the analysis.

3.4 Evaluation and Case Studies

We considered the entire set of apps in the F-Droid repository, as well as the top 100 apps from each category of Google Play. The evaluation of SENTINEL was performed on the entire subset of apps that contained sensor listeners (a total of 709 apps). The static analysis identified 18 apps for which the code exhibited the sensor leak patterns described earlier. Table 3.1 shows measurements for these apps. The first six apps are from F-Droid (also available at https://presto-osu.github.io/Sentinel) and the rest are from Google Play. Column "Class" shows the number of classes in the app. This number includes classes in libraries that are included in the

Table 3.1: Applications, paths, and tests.

	App size				L paths		$L \cap R$ paths		Tests		Leaks		Time
Application	Class	Stmt	Node	Edge	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2	(sec)
`Mtpms`	37	3148	20	38	7	8	6	6	1	1	1	1	0.1
`Drismo`	325	24592	425	645	1740	719	320	512	1	1	0	1	1.2
`Geopaparazzi`	1467	149469	283	534	615	721	24	24	1	1	1	1	2.6
`Itlogger`	296	30516	30	77	24	44	7	24	1	1	1	1	0.6
`AIMSICD`	921	79438	59	137	73	72	4	2	1	1	1	1	0.8
`Coregame`	44	1988	3	7	2	2	2	2	1	1	1	1	0.1
`NightVisionCamera`	408	44399	74	93	5	25	0	16	0	1	0	–	0.6
`Voxofon`	2637	184406	451	1084	4011	994	0	2	0	1	0	1	7.4
`VRVideoPlayer`	334	24296	13	29	5	8	2	2	1	1	0	0	0.5
`MobinCube`	502	67186	975	1165	12735	83209	3155	3907	3	5	0	0	16.9
`CSipSimple`	1319	111659	57	223	750	402	0	100	0	1	0	1	9.4
`Calculator Vault`	2025	137364	220	694	49064	13837	1128	3108	2	2	2	2	137.7
`Comebacks`	160	21246	67	96	46	50	0	16	0	1	0	–	0.3
`Pushups`	956	87545	627	1401	13731	2364	0	2	0	1	0	0	2.9
`Dogwhistier`	2415	181626	125	302	2257	1739	25	27	1	1	1	1	48.1
`Hideitpro`	3315	227087	807	1610	5171	1460	1058	1347	2	2	2	2	10.5
`LikeThatGarden`	1678	115092	634	1748	575773	201332	8165	21425	1	1	1	1	72.2
`MyMercy`	907	74914	258	629	286	400	0	5	0	1	0	–	11.7

app. Our analysis considers the code in all these classes and makes no distinction between app code and code in third-party libraries. Column "Stmt" contains the number of Soot IR statements for these classes. Columns "Node" and "Edge" show the total number of SG nodes and edges, respectively.

The next six columns show measurements for the number of SG paths. Under "L paths" are included the number of paths with matching open and close symbols—that is, paths from languages L_1 and L_2 (Section 3.2.2) limited by parameter $k = 4$ and without duplicated edges (Section 3.2.4). For many applications, the number of such paths is in the thousands. Executing test cases for each such path may be expensive. However, it is possible to reduce this number significantly by performing our static sensor analysis. The analysis identifies GUI event handlers and lifecycle callbacks

that trigger acquire and release symbols; based on this, it determines L_1 or L_2 paths that exhibit the sensor leak patterns. Columns "$L \cap R$ paths" show the numbers of paths that match the leak patterns. Clearly, significant reduction in the number of paths can be achieved. For further reduction, we use three filtering techniques to select the "guilty" window and to choose minimal-length paths (Section 3.2.4). Columns "Tests" shows the actual number of test cases generated by SENTINEL after this filtering. Again, significant reduction is observed, ultimately producing only a few test cases per app. These measurements demonstrate that static analysis of app code can successfully identify only a small subset of possible GUI event sequences that need to be executed at run time.

Columns "Leaks" show the number of executed test cases that resulted in an observed run-time leak. It is a common practice that an application has a template activity and some other activities are subclasses of this activity. If this parent activity has a defect, all of its subclasses will have the same defect. Therefore, when we report the results for columns "Tests" and "Leaks", we exclude test cases and leaks caused by the subclasses of the same defective parent activity class. Columns with "–" represent test cases that could not be executed, as described shortly. As can be expected, not every executed test case leads to leaking behavior, due to the conservative nature of static analysis. For 12 apps, the test cases exposed sensor leaks. Later we discuss examples of test cases that did not have leaks. It is worth noting that the apps listed in the table are not "toy" projects: in particular, the apps from Google Play are among the most popular in their categories and have many thousands of downloads from users. These results show that even popular applications can contain sensor leaks and our test generation approach can expose these leaks successfully.

Test generation time, in seconds, is shown in column "Time". It includes callback analysis of acquire and release effects, path checking, and test case generation. These measurements indicate that the cost of code analysis and test generation is practical.

Non-executable test cases. Three generated test cases could not be executed ("–" table entries). Our tests use an explicit intent to open the first activity in a test case. This is a typical approach for unit testing for Android, but in those three cases the activity crashes when opened. We also attempted, unsuccessfully, to trigger these activities using GUI sequences that start from the main app activity. For `MyMercy`, such a sequence requires a pre-existing medical account, which we are not able to obtain. For `NightVisionCamera` and `Comebacks`, the problematic activity is supposed to display a full-screen ad when the user clicks on an ad banner, but we were unable to trigger these ads on our device or in the emulator.

The results in Table 3.1 differ slightly from the ones in an earlier published version of this work [86]. Some differences are due to minor changes in the underlying GATOR tool (e.g., revised handling of ⟨activity-alias⟩ tags in XML files), and one difference is due to an inaccuracy in our manual investigation of run-time leaks.

3.5 Case Studies

This section briefly presents several case studies of apps described in Table 3.1. To understand the underlying reasons for the analysis reports, we examined manually the app code. For F-Droid apps, we examined the publicly-available source code. For Google Play apps, we used the `jadx` DEX-to-Java decompiler to study the app code.

Calculator Vault This app was already discussed; the relevant source code is presented in Figure 3.1. In the graph from Figure 3.2, the path with labels $\mathsf{open}(w_3)$, $\mathsf{acquire}(s), \mathsf{close}(w_3)$ is reported as a $P_1(w_3, s)$ path. We verified this leak on our Pixel phone. When the widget referenced by variable `sc` was clicked, the accelerometer sensor was enabled. This sensor remained turned on even after we exited both `UnlockActivity` and `SettingActivity` by clicking the BACK button: the Android debug bridge showed that the listener was still listening to changes from the sensor, which was clearly a leak.

CSipSimple This VoIP app was also discussed earlier and was illustrated in Figure 3.4. Activity `InCallActivity` will be started by an Android intent broadcast when there is an incoming or an outgoing call. When this activity is launched, lifecycle callback `onCreate` will be invoked and its callee method `startTracking` will acquire the proximity sensor. The activity does release the sensor in `stopTracking`, which is invoked by callback `onDestroy`. However, if a user presses the HOME button during the call and navigates to other applications, e.g. browsing a web page or looking up a contact, the acquired sensor will still be held by `InCallActivity` even though it is not responding to user interactions. The corresponding P_2 path was reported by the static analysis and was verified on the Pixel phone used in our experiments.

Geopaparazzi This F-Droid app, which is also available in Google Play, is used for engineering and geologic surveys. Figure 3.7 shows the simplified code for the leak.

```
1  class SensorManagerL implements
2      SensorEventListener {
3    SensorManagerL sml;
4    SensorManager sm;
5    static SensorManagerL getInstance(...) {
6      if (sml == null) {
7        sml = new SensorManagerL();
8        sm = (SensorManager)
9             getSystemService(SENSOR_SERVICE);
10       sml.startSensorListening();}
11     return sml; }
12   void startSensorListening() {
13     Sensor accel = sm.getDefaultSensor(
14       Sensor.TYPE_ACCELEROMETER);
15     sm.registerListener(sml, accel); ... }
16   void stopSensorListening() {
17     sm.unregisterListener(sml);}
18   void onSensorChanged(...) {...} }
19 class GeoPaparazziActivity extends Activity {
20   SensorManagerL sml;
21   void onCreate(...) { init(); ... }
22   void init() {
23      sml = SensorManagerL.getInstance(); } }
```

Figure 3.7: Example derived from Geopaparazzi.

The app uses a wrapper class `SensorManagerL` to process all sensor-related operations. (We use this name for brevity; in reality, this is app class `eu.hydrologis.geopaparazzi.SensorManager`.) The class implements the singleton pattern. At line 15, method `startSensorListening` registers a listener for the accelerometer. This method is called during the singleton object creation (line 10). Callback method `onCreate` of the `GeoPaparazziActivity` calls `init`, which instantiates the singleton and acquires the sensor. The only method that releases the sensor is `stopSensorListening` (line 16). However, this method is not called by any app component. Once the activity turns on the accelerometer sensor, it can only be turned off by killing the

app. We have verified this behavior in our test platform and confirmed that it leads to a sensor leak.

mTpms This app from F-Droid is a motorcycle tire pressure monitor system reader. It uses the device's light sensor to detect changes of ambient light. It will change the background and text to dark colors when it detects that the level of ambient light is below a certain threshold. In the `onCreate` method of the main activity, the app obtains the sensor object for the light sensor and registers a listener for it. However, there is no app code that unregisters this listener. Therefore, the light sensor will be turned on when this application is launched from the Android Launcher and will remain on unless this application is killed. This behavior was confirmed on our Pixel phone.

False positives The false positives shown in Table 3.1 are in apps `Mobincube`, `Pushups`, `VillageVR`, and `Drismo`. In the first three cases, there are classes containing methods which override the same methods in their superclasses. The subclass methods acquire sensors and leak them. However, these subclasses are never instantiated to objects at run time. Due to the use of class hierarchy analysis, when our analysis encounters an invocation of the superclass method, it incorrectly determines that the called method could be from the defective never-instantiated subclass. This imprecision causes the false positives. This is a well-known limitation of class hierarchy analysis [20]. There is wide variety of options for more precise call graph construction for object-oriented languages [32, 75, 79] and using some of these techniques would eliminate these false positives. One could argue that such code is a "time bomb" that

could affect future versions of the app, and therefore should be reported and eliminated. However, since we were unable to trigger the leaks at run time, we counted these cases as false positives. The last case, the false positive in `Drismo` is caused by a limitation in the WTG construction of GATOR. In `Drismo`, the defective activity acquires a sensor resource in `onCreate` and releases it in the `onBackPressed` callback, which is the event handler callback for the hardware BACK button. However, the `onBackPressed` callback is not considered in GATOR. Therefore, SENTINEL cannot detect the release effects in this `onBackPressed` callback, causing a false positive. If this WTG limitation is eliminated, this false positive will be eliminated as well.

3.6 Conclusions

This work demonstrates that sensor leaks in Android apps can be naturally expressed as CFL-R properties and it is possible to automatically generate effective tests for sensor leaks in Android apps. This machinery models the interleaving of two important aspects of app behavior: UI changes due to opening and closing of windows, and sensor changes due to registration and de-registration of listeners. Both leaks beyond window lifetime and leaks in suspended state are easy to express in this model. Exploration of CFL paths can be done efficiently and can produce useful leak reports. Experimental studies indicate that analysis precision is high and that sensor leaks in realistic Android apps can be successfully detected.

CHAPTER 4: Refactoring of Energy-Inefficient Scheduled Tasks

In this chapter, we focus on energy inefficiencies due to the improper usage of Android's `AlarmManager` service. `AlarmManager` is an Android system service that allows an application to schedule an alarm (at a fixed time) or repeating alarms (at fixed time intervals). This service is widely used by applications to perform tasks such as checking for updates and uploading application data to remote servers. However, because such alarms are scheduled at fixed times, they may cause excessive device wake ups which are energy-inefficient. Because of this, in current Android releases, `AlarmManager` is no longer recommended for this type of tasks. Starting from Android 8.0, using `AlarmManager` to start a background service is no longer allowed. Instead, the recommendation to developers is to use the `SyncManager` or `JobService` system services. Due to legacy reasons, there are still large numbers of applications using `AlarmManager` for scheduled tasks, which may cause energy drain on user's devices.

We propose a static analysis to detect the patterns of usage of `AlarmManager`. The results of this analysis are then used by our code refactoring component to automatically patch the application to use `JobService`. By measuring the power consumption of original applications and refactored applications, we demonstrate that this approach can help reduce the energy consumption of these applications.

4.1 Background

4.1.1 Relevant Features of Android's AlarmManager

This section describes the relevant features of Android's `AlarmManager` system service. It is a common practice that Android apps perform periodic tasks in the background. For example, a weather app may update the current weather every hour, and an e-mail app may synchronize user's mailbox every 15 minutes. For these types of tasks, early versions of Android include the `AlarmManager` system service, which provides similar features to timers in modern desktop OS. Figure 4.1 shows an example of an application using `AlarmManager`.

Intent and PendingIntent Before discussing the usage of `AlarmManager`, we need to consider Android's `Intent` and `PendingIntent` features, which are essential parts of inter-component communication. According to the Android development reference [6], an `Intent` is an abstract data structure used to describe the operation that should be performed. This information can be used, for example, in APIs such as `startActivity` and `startService`, which launch new activities and services, respectively. An `Intent` is either explicit or implicit. An explicit `Intent` contains an explicit description of the component to be launched—for example, the specific activity class to be run. An implicit `Intent` does not contain this information; instead, it must have other information such as the *action* and *category* for the operation, which allows the system to determine the best available component for this operation. When scheduling an alarm using `AlarmManager`, only explicit `Intent` will be accepted. Line 13 in Figure 4.1 shows an example of creation of an explicit `Intent`, which has a target component `SyncAlarmReceiver`.

A `PendingIntent` is a wrapper around an `Intent` and its target action. It is used when the application needs to send a request to system services or another application to perform an action at a later time. For example, when an application schedules an alarm using `AlarmManager`, a `PendingIntent` object is required. When the alarm is due, `AlarmManager` will perform the action defined by this `PendingIntent`. Lines 14–15 in Figure 4.1 show the creation of a `PendingIntent` object which wraps the `Intent` object created at line 13, which defines the action of sending a broadcast to `SyncAlarmReceiver`.

BroadcastReceiver In an Android app, an alarm is usually used to trigger a `BroadcastReceiver` or a `Service`. A callback method `onReceive` is defined in a subclass of `BroadcastReceiver`. This method will be invoked by the Android framework when this broadcast receiver is triggered. Due to its simplicity, this is the most common usage of `AlarmManager`. Line 18 in Figure 4.1 shows an example of a broadcast receiver.

Compared to `BroadcastReceiver`, a `Service` is more complex, as it has its own lifecycle and interfaces for inter-component communication. When `Service` is used in an alarm, a subclass `IntentService` is commonly used by developers. A subclass of `IntentService` contains a callback method `onHandleIntent`, which will be invoked by the Android framework when an `Intent` is fired to this `IntentService`.

4.1.2 AlarmManager

Figure 4.1 shows an example of `AlarmManager` usage. Application Moloko is a mobile client application for web site `rememberthemilk.com` [72] which maintains user's to-do lists. The app has over 1 million installs from the Google Play store.

```
1 class AlarmManagerPeriodicSyncHandler {
2  void setPeriodicSync(long startUtc, long intervalMs) {
3    AlarmManager aM = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);
4    long nowUtc = System.currentTimeMillis();
5    if (startUtc < nowUtc ) {
6      startUtc = nowUtc;
7      PendingIntent syncIntent = createSyncAlarmIntent(context);
8      aM.setRepeating(RTC_WAKEUP, startUtc,
9                                intervalMs, syncIntent);
10    }
11 }
12 PendingIntent createSyncAlarmIntent(Context context) {
13    Intent intent = new Intent(context, SyncAlarmReceiver.class);
14    return PendingIntent.getBroadcast(context, 0, intent,
15                         PendingIntent.FLAG_UPDATE_CURRENT );
16   }
17 }
18 class SyncAlarmReceiver extends BroadcastReceiver {
19   void onReceive(Context context, Intent intent) {
20     final Account account = AccountUtils.getRtmAccount(
21         context.getApplicationContext());
22     if (account != null) {
23       if (SyncUtils.isReadyToSync(context.getApplicationContext())) {
24         SyncUtils.requestScheduledSync( account );
25       }
26     } else {
28       MolokoApp.get( context.getApplicationContext() ).stopPeriodicSync();
29     }
30   }
31 }
```

Figure 4.1: Example derived from the Moloko application.

Method `setPeriodicSync` retrieves an instance of `AlarmManager` (line 3) and then creates an intent which points to the broadcast receiver `SyncAlarmReceiver` (line 13). A pending intent is created for this intent and is used to schedule a repeating alarm using API `setRepeating` at lines 7–9. Argument `RTC_WAKEUP` means the timer is using the real time clock in milliseconds as reference. This alarm will wake up the device if necessary. Argument `startUtc` defines the time in real time clock when this alarm should be fired in the first time. Argument `intervalMs` defines the milliseconds of repeating interval. In this example, callback `onReceive` in `SyncAlarmReceiver` will be invoked every `intervalMs` milliseconds.

4.1.3 Energy Impact

Smartphones such as Android devices have limited battery capacity. To reduce energy consumption, when the screen is off, Android will aggressively put the CPU into sleep state, which pauses execution on all CPU cores. When scheduling an alarm using `AlarmManager`, developer can specify the type of the alarm. There are four types available: RTC, RTC_WAKEUP, ELAPSED_REALTIME, and ELAPSED_REALTIME_WAKEUP. The prefix "RTC" means the the supplied delay is using current real time clock as reference, which can be retrieved using API `System.currentTimeMillis()`. The other prefix "ELAPSED_REALTIME" means the time since device boot up will be used as reference, which can be retrieved using `SystemClock.elapsedRealtime()`. The type name with "WAKEUP" suffix means that these alarms will wake up the device if the device CPU is in sleep state. The other two types, RTC and ELAPSED_REALTIME, will not wake up the device. Instead, they will be fired when the device is woken by other events. The energy consumed by executing these alarms are related to the CPU time of each alarm and the frequency with which each alarm is fired. If several alarms with "WAKEUP" are fired within a small time window, they will consume less energy than the case if they are fired one by one with significant time intervals between them.

Figure 4.2 illustrates frequent device wake ups in two scenarios. A red bar in this figure indicates the device is woken by an alarm and is executing a background task. The upper part of the figure represents a device with multiple apps, where each app schedules its alarms at an arbitrary time or time interval. The bottom part of the figure represents a device with same set of apps installed; however, the alarms are ideally scheduled in small time windows. In this case, the background tasks are

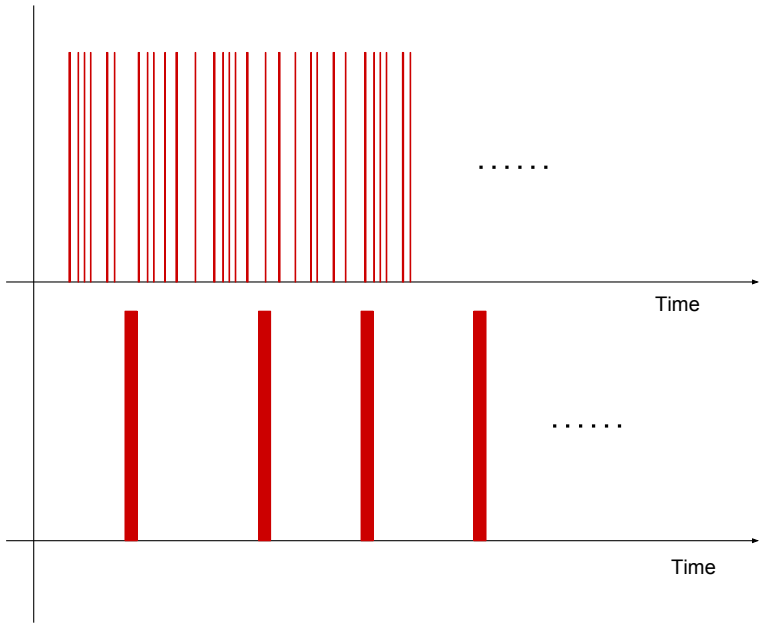

Figure 4.2: Illustration of different wake up patterns

executed together. On an Android device, the second case will consume less energy compared to the first one.

There are multiple factors that contribute to differences in energy consumption. When tasks are scheduled together (batched) in the second case, the total execution time of the CPU is smaller than the first case because most CPUs used in mobile devices are multi-cores, which can process multiple tasks at the same time. The smaller execution time results in less energy consumption. For periodic background tasks, it is common to see these tasks use hardware resources such as device sensors/GPS, or transferring data over the network. These hardware resources exhibit a tail energy

effect. When one of these types of resources is requested and released, the hardware will keep consuming energy for a short time before being completely turned off. This behavior has been discussed in prior work [10, 36, 46]. In this case, batching background tasks in a short window will help reduce energy consumption.

4.1.4 Android JobService

In Android 5.0, Google introduced `JobService`, a subclass of `Service`, which is better suited for background tasks. A `JobService` defines a job that can be scheduled to be executed in the background, using system service `JobScheduler`. Besides the start delay and repeating interval which can be specified for alarms with `AlarmManager`, `JobService` can be scheduled with other requirements such as requiring Wi-Fi connection or expecting that the device is charging using AC power. When the requirements of a job are satisfied, the system will execute it. As a result, the corresponding callbacks defined in class `JobService` will be invoked. Compared to alarms managed by `AlarmManager`, jobs are no longer executed at fixed times. Android will try to batch and defer executing the jobs, in order to preserve energy. In the official Android documentation, `JobService` is recommended for background tasks instead of the traditional `BroadcastReceiver` with `AlarmManager` [84].

Figure 4.3 shows an example of `JobService` usage derived from the Muzei app. We use this app to illustrate how this Android feature could be used in app code. `DownloadArtworkJobService` is a subclass of `JobService`. Its purpose is updating the current wallpaper every 15 minutes. It is scheduled for execution in convenience method `schedule`. This method retrieves an instance of `JobScheduler` (line 5). To

```
1 public class DownloadArtworkJobService extends JobService {
2   DownloadArtworkTask mTask = null;
3   static final int JOB_ID = 100;
4   static schedule(Context context) {
5     JobScheduler js = context.getSystemService(Context.JOB_SCHEDULER_SERVICE);
6     JobInfo.Builder builder = new JobInfo.Builder(JOB_ID,
7         new ComponentName(context, DownloadArtworkJobService.class))
8         .setPeriodic(15*60*1000);
9     JobInfo info = builder.build();
10    js.schedule(info);
11  }
12  @Override
13  boolean onStartJob(final JobParameters params) {
14    mTask = new DownloadArtworkTask(this) {
15    @Override
16      protected void onPostExecute(Boolean success) {
17        super.onPostExecute(success);
18        jobFinished(params, !success);
19      }};
20    mTask.execute();
21    return true;
22  }
23  @Override
24  boolean onStopJob(final JobParameters params) {
25    if (mTask != null) {
26      mTask.cancel(true);
27    }
28    return true;
29  }
30 }
```

Figure 4.3: Example derived from the Muzei application.

specify the requirements of the job, a `JobInfo` object which describes such requirements is required. Lines 6–9 shows an example of a `JobInfo` object. Line 7 specifies a request to create an instance of class `DownloadArtworkJobService` and to start it (i.e., to invoke callback `onStartJob` on it). Line 8 defines that this creation/start action should be repeated every 15 minutes. In the running example, the job is scheduled by calling the `schedule` API from `JobScheduler` (line 10).

Each `JobService` class contains two callbacks: `onStartJob` (line 13) and `onStopJob` (line 24). Callback `onStartJob` will be invoked when the job is beginning execution and should contain code that performs the actual background tasks. Similarly to

the `onReceive` callback in a `BroadcastReceiver`, `onStartJob` is executed on application's UI thread, which is not suitable for long-running tasks. It is recommended to offload such tasks to a second thread and to return `true` from `onStartJob`. This return value indicates to the underlying Android framework that the job has not yet completed execution. The system will then hold a wake lock on behalf of this app to allow tasks on other threads to be executed. The application can call API `jobFinished` to notify the system that the job has finished execution. The system will then remove the wake lock.

If a job is running for too long, or its requirements are not longer satisfied (e.g, being plugged to AC power is no longer true), callback `onStopJob` will be called to notify the application that the job should be stopped. The code for this callback should stop the tasks that are currently executing. For short tasks that can be safely executed on the UI thread, the developer can put them within the `onStartJob` method and return `false` from this method. This notifies the system that the job has finished and there is no need to acquire and hold a wake lock for this job.

In the running example, object `mTask` (line 14) is created in callback `onStartJob` as an instance of `DownloadArtworkTask` class. This class is a subclass of `AsyncTask`, an Android component that offloads task to other threads instead of the UI thread. Callback `onPostExecuted`, which will be invoked on the `DownloadArtworkTask` instance when its execution completes, in turn contains a call to `jobFinished` (line 18) to notify the system that the job is finished. Callback `onStopJob` contains a call to `cancel` to force the `DownloadArtworkTask` defined by `mTask` to stop executing. Compared to a `BroadcastReceiver`, `JobService` provides better wake locks management and is better suited for periodic background tasks.

4.1.5 Energy-Inefficient AlarmManager Patterns

In this work, we use following conditions to define an energy-inefficient usage of `AlarmManager`:

- The scheduled alarm will wake up the device
- The scheduled alarm has fixed triggering time or has fixed repeating interval, which does not give system the flexibility to batch the alarms

If a device has multiple applications that satisfy these conditions, the device will likely not be put into sleep for a long time. Section 4.5 presents experimental results that shows this behavior on a real device.

In a preliminary study, we manually examined the source code of 82 open source apps that use `AlarmManager`. From this study, we concluded that apps that use `BroadcastReceivers` to get alarms from `AlarmManager` can be feasibly refactored automatically using static code analysis and rewriting. While some apps do use `Services` (i.e. `IntentServices`) to get alarms with `AlarmManager`, their more complicated lifecycle makes automated refactoring infeasible. Therefore, we decided to target the following pattern of `AlarmManager`: (1) an application uses either RTC_WAKEUP or ELAPSED_REALTIME_WAKEUP when scheduling an alarm with `AlarmManager`; (2) the APIs used in alarm scheduling are `set`, `setExact`, or `setRepeating`, which prevent the system from batching the alarms; (3) the target of the scheduled alarm is a `BroadcastReceiver`.

When an application has an alarm scheduled in this patten, this may cause energy inefficiency. In this work, we designed a static analysis to detect instances of this pattern, as well as a code rewriting component to perform automated code refactoring

to convert these `AlarmManager` usages to `JobService` usages. The evaluation of our approach shows that it reduces device wake ups and improves battery consumptions.

4.2 Semantics of Relevant Android Constructs

In order to design static analyses targeted at the energy-inefficiency patterns discussed earlier, the run-time semantics of Android `AlarmManager` usage is necessary. In this section, we present the definition of these semantics.

4.2.1 Plain Java and Plain Android

We define the run-time semantics of Android `AlarmManager` usage based on similar semantic definitions for "plain" Java and "plain" Android, which are derived from prior work [63, 74, 92].

Plain Java As our main focus is the semantics of individual statements of a method body, we elide irrelevant language features such as the modeling of the type system. These features are well understood and have been explained in prior work.

A Java program consists of a set of classes; "classes" here refers to both classes and interfaces. Each class defines fields, methods, and constructors. The fields are denoted by $f \in \mathsf{Fld}$. The body of a method contains local variables $x \in \mathsf{Var}$ and statements. The syntax of statements can be defined by:

$$s ::= \quad x := \texttt{new}\ c \mid x := y \mid x := y.f \mid x.f := y$$

As typically done in reference analysis for Java, we omit conditional statements and loops. We use set Obj to represent heap objects, map Env to define the relationship between local variables and these heap objects, and map Heap to represent the

object fields' values. Figure 4.4 illustrates the domains and functions used to define the semantics of plain Java.

$$
\begin{array}{lll}
o \in \mathsf{Obj} & & \text{heap objects} \\
\nu \in \mathsf{Env} = \mathsf{Var} \to \mathsf{Obj} & & \text{environments} \\
\mathcal{P} \in \mathsf{Heap} = \mathsf{Obj} \times \mathsf{Fld} \to \mathsf{Obj} & & \text{heaps}
\end{array}
$$

$$
\begin{array}{rcl}
\langle \mathcal{P}, \nu, x := \texttt{new } C \rangle & \to & \langle \mathcal{P}, \nu[x \mapsto o] \rangle \\
\langle \mathcal{P}, \nu, x := y \rangle & \to & \langle \mathcal{P}, \nu[x \mapsto \nu(y)] \rangle \\
\langle \mathcal{P}, \nu, x := y.f \rangle & \to & \langle \mathcal{P}, \nu[x \mapsto \mathcal{P}(\nu(y), f)] \rangle \\
\langle \mathcal{P}, \nu, x.f := y \rangle & \to & \langle \mathcal{P}[(\nu(x), f) \mapsto \nu(y)], \nu \rangle
\end{array}
$$

Figure 4.4: Semantic domains and functions

The rules show the updates to Env and Heap due to different statements. The notation of $\nu[x \mapsto o]$ means that x is (re)mapped to o in the map ν. In the rule for $x :=$ `new` C, o represents that a new heap object $o \in \mathsf{Obj}$ is created, and this object is an instance of class C.

Plain Android Prior work [74, 92] defined the semantics of Android GUIs and features like activities, menus, etc. These definitions are not directly related to the `AlarmManager` usage, however, we use their definition as basis of our semantics. The semantics of Android `AlarmManager` usage is described below.

4.2.2 Semantics of AlarmManager

Intent and PendingIntent

As explained in Section 4.1.1, applications use instances of `PendingIntent` to specify alarm targets. Each `PendingIntent` warps an instance of `Intent`. We use a semantic definition for `Intent` and `PendingIntent` derived from prior work [96].

Instances of `Intent` and `PendingIntent` and the sets of all such instances are denoted as follows

$$\begin{array}{lll} in & \in \mathsf{Intent} \subset \mathsf{Obj} & \text{intents} \\ pi & \in \mathsf{PendingIntent} \subset \mathsf{Obj} & \text{pendingintents} \end{array}$$

There are several categories of API calls related to `Intent` and `PendingIntent` creation and manipulation. For `AlarmManager` usages, we only consider a subset of these APIs, as defined by the following abstract syntax:

$$s ::= \ldots \; x := \texttt{new Intent}(y) \mid x := \texttt{buildpending}(y)$$

Statement $x := \texttt{new Intent}(y)$ represents that a new instance $in \in \mathsf{Intent}$ is created and assigned 'to variable x. Because only explicit `Intent`s can be used in `AlarmManager`, we only consider this constructor for `Intent` as it is the only way to create an explicit `Intent`. As shown at line 13 in Figure 4.1, parameter y is a Java `Class` object (or an equivalent `String` object). This target class is a `BroadcastReceiver`, a `Service`, or an `Activity`. In the rest of this chapter, we use Class to represent the set of all `Class` objects. Statement $x := \texttt{buildpending}(y)$ represents the creation of a new `PendingIntent`. Here y refers to an instance of `Intent`. Operation `buildpending` returns the newly constructed instance of `PendingIntent`. To express the semantics, we extend the definition of heap as follows:

$$\mathsf{Heap} = \ldots \cup (\mathsf{Intent} \times \{\texttt{target}\} \to \mathsf{Class}) \cup (\mathsf{PendingIntent} \times \{\texttt{target}\} \to \mathsf{Intent})$$

The artificial field $\texttt{target} \in \mathsf{Fld}$ of an `Intent` instance represents its target class. The artificial field $\texttt{target} \in \mathsf{Fld}$ of a `PendingIntent` instance represents the warped `Intent` instance. The semantic effects of the relevant statements are:

$$[\textsc{NewIntent}] \quad \langle \mathcal{P}, \nu, x := \texttt{new Intent}(y) \rangle \rightarrow \langle \mathcal{P}[(o, \texttt{target}) \mapsto \nu(y)], \nu[x \mapsto o] \rangle$$
$$[\textsc{BuildPending}] \quad \langle \mathcal{P}, \nu, x := \texttt{buildpending}(y) \rangle \rightarrow \langle \mathcal{P}[(o, \texttt{target}) \mapsto \nu(y)], \nu[x \mapsto o] \rangle$$

Here o denotes a new $o \in \mathsf{Obj}$ that is created as a result of the operation.

Scheduling an Alarm

When scheduling an alarm using `AlarmManager`, the type of alarm, start delay, and alarm target need to be provided. If it is a repeating alarm, the time interval is also needed. To help ease the design of static analysis, we introduce an artificial class `Alarm` in our semantic definitions. For each invocation to `AlarmManager` scheduling method, a new instance of this class is artificially created. In fact, a similar object will be created "behind the scenes" in the system framework at run time. Instances of `Alarm` and the set of all such instances are denoted by

$$ao \in \mathsf{Alarm} \subset \mathsf{Obj} \text{ artificial alarm object}$$

We also extend the definition of the heap as follows:

$$\begin{aligned} \mathsf{Heap} = \ldots \cup \\ & (\mathsf{Alarm} \times \{\texttt{type}\} \rightarrow \mathsf{Int}) \cup \\ & (\mathsf{Alarm} \times \{\texttt{delay}\} \rightarrow \mathsf{Int}) \cup \\ & (\mathsf{Alarm} \times \{\texttt{interval}\} \rightarrow \mathsf{Int}) \cup \\ & (\mathsf{Alarm} \times \{\texttt{pending}\} \rightarrow \mathsf{PendingIntent}) \end{aligned}$$

Here `type`, `delay`, `interval` and `pending` are four artificial fields of a `Alarm` object. Field `type` represents the type of the alarm, which should be an integer with value from the four integer constants defined in class `AlarmManager`: `RTC`, `RTC_WAKEUP`, `ELAPSED_REALTIME`, and `ELAPSED_REALTIME_WAKEUP`. Fields `delay` and `interval` also have integer values. They represent the start delay of the scheduled alarm, and an

interval for a reoccurring alarm, respectively. For alarms only occur once, this field will be 0. Field `pending` refers to an instance of `PendingIntent` that represents the target of the alarm.

There are two categories of APIs for alarm scheduling: ones that schedule a single alarm and ones that schedule repeating alarms. We use abstract operation `set` to represent both `set` and `setExact` APIs and use `setRepeaing` to represent `setRepeating` API. The abstract syntax can be defined as follows:

$$s ::= \ldots \mid \texttt{set}(t, d, p) \mid \texttt{setRepeating}(t, d, i, p)$$

Here each operation `set` and `setRepeating` implicitly creates an instance of an artificial `Alarm` class. We only use these artificial instances for defining the abstracted semantics and the static analysis based on it. Parameter t refers to the type of the alarm, which can be one of four integer constants mentioned earlier. We use set $\mathsf{AlarmType} \subset \mathsf{Int}$ to represent the set of possible type values. Parameter d refers to the start delay of the alarm and parameter i refers to the time interval for repeating alarms. Both of these two parameters are integers, denoted by set Int. Parameter p refers to the target of the alarm, which is an element of set $\mathsf{PendingIntent}$. The semantic effects for these operations are defined as follows:

$$\begin{array}{rrl}
[\textsc{Set}] & \langle \mathcal{P}, \nu, \texttt{set}(t, d, p) \rangle \rightarrow & \langle \mathcal{P}[(o, \texttt{type}) \mapsto \nu(t), \\
 & & (o, \texttt{delay}) \mapsto \nu(d), \\
 & & (o, \texttt{pending}) \mapsto \nu(p)], \nu \rangle \\
[\textsc{SetRepeating}] & \langle \mathcal{P}, \nu, \texttt{setRepeating}(t, d, i, p) \rangle \rightarrow & \langle \mathcal{P}[(o, \texttt{type}) \mapsto \nu(t), \\
 & & (o, \texttt{delay}) \mapsto \nu(d), \\
 & & (o, \texttt{interval}) \mapsto \nu(i), \\
 & & (o, \texttt{pending}) \mapsto \nu(p)], \nu \rangle
\end{array}$$

Here o denotes a new $o \in \mathsf{Alarm}$ that is created as a result of the operation.

4.3 Static Reference Analysis

In this section, we illustrate the static analyses we designed to detect inefficient `AlarmManager` usage. The first part of the description illustrates a static analysis that models the propagation of `AlarmManager` related objects. The second part illustrates the defect detection based on the results of reference analysis and callback sequence traversals. Both of these analyses are based on the three-address Jimple representation used in the Soot static analysis framework [77].

Given the semantics defined in Section 4.2, we aim to develop static reference analysis for creation and propagation of `Intent`, `PendingIntent` as well as the artificial `Alarm` object. For the simplified semantics of plain Java, the reference analysis can be solved by using a *constraint graph*. Each node in the graph corresponds to a local variable $x \in \mathsf{Var}$, a field $f \in \mathsf{Fld}$ or an object allocation `new` C. Each edge represents the constraints on values. For an assignment statement $x := y$, there is an edge $y \rightarrow x$, which means the value of x contains the value set of y. Similarly, for an object allocation statement $x :=$ `new` C, there is an edge `new` $C \rightarrow x$, which means the value of x contains the allocated object instance of C. Such an analysis is usually classified as flow-insensitive, context-insensitive and field-based analysis [43, 75]. Our analysis of alarms is an extension of this standard analysis.

Figure 4.5 shows the conceptual input to the analysis for the running example shown in Figure 4.1. This program representation is based on the abstract semantics presented earlier.

```
1 long nowUtc = System.getCurrentTimeMillis();
2 startUtc = nowUtc;
3 c = SyncAlarmReceiver.class
4 Intent i = new Intent(c)
5 PendingIntent syncIntent = buildpending(i)
6 t = RTC_WAKEUP;
7 setRepeating(t, startUtc, intervalMs, syncIntent)
```

Figure 4.5: Abstract program representation.

4.3.1 Constraint Graph

Figure 4.6 shows the constraint graph of the running example based on the program representation from Figure 4.5. Some nodes are postfixed with numbers which correspond to line numbers in Figure 4.1.

Operation Nodes We use the term operation nodes to denote the nodes that represent abstract actions, due to statement of the form $x := \mathtt{op}(y)$. An operation node will have an incoming edge from the node of parameter y; for actions that take more than one parameter, there will be a separate edge for each parameter. If there is a return value for the operation, and this value is assigned to x, there is a corresponding outgoing edge to the node for x. Let OP denote the set of such operation nodes.

Object Nodes `Intent`, `PendingIntent` and `Alarm` objects are created in program statements. Let IN be the set of `Intent` nodes created by $x :=$ `new Intent`(y), PI be the set of `PendingIntent` nodes created by $x :=$ `buildpending`(y), and AM be the

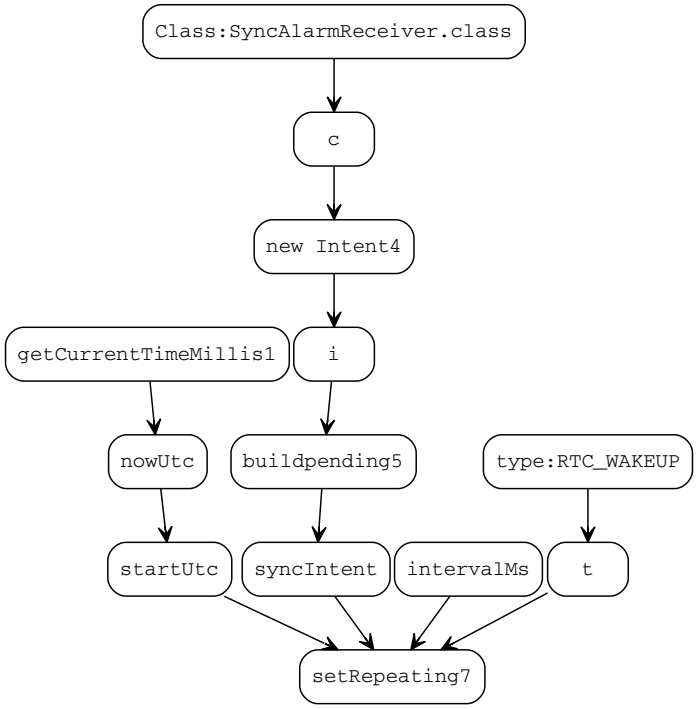

Figure 4.6: Example of a constraint graph

set of `Alarm` nodes. We will also we use INT to represent integer nodes and CL to represent Class object nodes.

4.3.2 Constraint Analysis

In this subsection, we define several inference rules, which are the basis for the reference analysis. Relation *flowsto* $\subseteq \{\mathsf{IN} \cup \mathsf{PI} \cup \mathsf{INT} \cup \mathsf{CL}\} \times \{\mathsf{Var} \cup \mathsf{Fld} \cup \mathsf{OP}\}$ represents the flow of objects and integer values to local variables, fields, and operation nodes. The inference rules for this relation are

$$\frac{n_1 \in \mathsf{IN} \cup \mathsf{PI} \cup \mathsf{INT} \cup \mathsf{CL} \qquad n_2 \in \mathsf{Var} \qquad n_1 \rightarrow n_2}{n_1 \;\; \textit{flowsto} \;\; n_2}$$

$$\frac{n_2 \in \mathsf{Var} \cup \mathsf{Fld} \qquad n_3 \in \mathsf{Var} \cup \mathsf{Fld} \cup \mathsf{OP} \qquad n_2 \rightarrow n_3 \qquad n_1 \;\; \mathit{flowsto} \;\; n_2}{n_1 \;\; \mathit{flowsto} \;\; n_3}$$

For example, in Figure 4.6, `Class` constant `Class : SyncAlarmReceiver.class` flows to operation node `newIntent4` via node `c`, which represents a local variable. Type constant `type : RTC_WAKEUP` flows to operation node `setRepeating7` via variable node `t`.

Intent and PendingIntent As stated earlier, instances of explicit `Intent` and `PendingIntent` are created through operations. Both `Intent` and `PendingIntent` have an artificial field `target` which points to the parameter passed to the operation at object creation. To capture this behavior, we use relation $\mathit{targets} \subseteq (\mathsf{IN} \times \mathsf{CL}) \cup (\mathsf{PI} \times \mathsf{IN})$. For semantic rules [NewIntent] and [BuildPending], the inference rules are defined at follows:

$$\frac{c \in \mathsf{CL} \qquad c \;\; \mathit{flowsto} \;\; \mathit{newintent} \qquad \mathit{newintent} \rightarrow n}{i \in \mathsf{IN} \qquad i \;\; \mathit{flowsto} \;\; n \qquad i \;\; \mathit{targets} \;\; c}$$

$$\frac{i \in \mathsf{IN} \qquad i \;\; \mathit{flowsto} \;\; \mathit{buildpending} \qquad \mathit{buildpending} \rightarrow n}{p \in \mathsf{PI} \qquad p \;\; \mathit{flowsto} \;\; n \qquad p \;\; \mathit{targets} \;\; i}$$

Here i denotes a newly created `Intent` object and p denotes a newly created `PendingIntent` object.

Scheduling an Alarm Operations $\texttt{set}(t, d, p)$ and $\texttt{setRepeating}(t, d, i, p)$ will schedule alarms based on the provided parameter values. Nodes that flow to operation nodes *set* and *setRepeating* determine the behavior of the alarms. For a `PendingIntent` node, we would like to know the target `Class` object it points to.

Therefore, record the quadruple of type, delay, interval, and target as a static abstraction of a run-time alarm object. We use $\mathsf{AM} \in \mathsf{INT} \times \mathsf{INT} \times \mathsf{INT} \times \mathsf{CL}$ to denote the set of all recorded quadruples.

$$\frac{t_1 \;\; \mathit{flowsto} \;\; \mathit{set} \qquad d_2 \;\; \mathit{flowsto} \;\; \mathit{set} \qquad p_3 \;\; \mathit{flowsto} \;\; \mathit{set} \qquad p_3 \;\; \mathit{targets} \;\; \mathit{in} \qquad \mathit{in} \;\; \mathit{targets} \;\; \mathit{cl}}{(t_1, d_2, 0, \mathit{cl}) \in \mathsf{AM}}$$

$$\frac{\begin{array}{c} t_1 \;\; \mathit{flowsto} \;\; \mathit{setRepeating} \\ d_2 \;\; \mathit{flowsto} \;\; \mathit{setRepeating} \qquad i_3 \;\; \mathit{flowsto} \;\; \mathit{setRepeating} \\ p_4 \;\; \mathit{flowsto} \;\; \mathit{setRepeating} \qquad p_4 \;\; \mathit{targets} \;\; \mathit{in} \qquad \mathit{in} \;\; \mathit{targets} \;\; \mathit{cl} \end{array}}{(t_1, d_2, i_3, \mathit{cl}) \in \mathsf{AM}}$$

Analysis Algorithm The computation of a solution to these constraints is done in several steps. The first step is to construct the constraints graph from the application code. The second step is to compute the creation of nodes $n \in \mathsf{IN} \cup \mathsf{PI}$ and their reachability to *set* and *setRepeating* operation nodes. The third step is to determine the set AM.

4.3.3 Analysis Output

The purpose of the static analysis it to provide information for the subsequent automated code refactoring. For code refactoring, the scheduling of an energy-inefficient alarm can be determined by the tuples from set AM created at *set* and *setRepeating* operation nodes. The program statement for such an operation node is reported if (1) the alarm type is either `RTC_WAKEUP` or `ELAPSED_REALTIME_WAKEUP`, (2) the delay or interval is a fixed constant value, and (3) the target is a `BroadcastReceiver`. We output all quadruples in set AM, together with their creating alarm-scheduling statements. This output is supplied to the automated code refactoring step.

4.4 Automated Code Refactoring

The purpose of the proposed automated code refactoring is to reduce excessive device wake ups, by converting energy-inefficient alarms scheduled with `AlarmManager` into `JobServices` scheduled with `JobScheduler`. For a single application, the refactoring process consists of the following steps.

4.4.1 Instrument PendingIntent Creation

While the static analyses described in Section 4.3 can identify the correct target class of the `PendingIntent`, it is still necessary to retrieve and record the creation of every `PendingIntent` object at run time. There are two main reasons: (1) callback `onReceive` in `BroadcastReceiver` requires an instance of `Intent` as a parameter; (2) the developer may put customized information in an `Intent` to be used by the target class later, which cannot be reliably tracked by static data flow analysis. For `AlarmManager` usage, there are three APIs that can be used for `PendingIntent` object creation:

- `PendingIntent getBroadcast(Context, int, Intent, int)`
- `PendingIntent getService(Context, int, Intent, int)`
- `PendingIntent getActivity(Context, int, Intent, int)`

As we focus on refactoring energy-inefficient alarms targeting `BroadcastReceiver`, only `getBroadcast` is considered. The instrumentation process for the creation of `PendingIntent` instances is straightforward. We insert a new class named `PrestoJobServiceFramework` into the application code. In this class, we create a static method which has exactly the same signature as the `getBroadcast` API. We then maintain a

```
1 static PendingIntent getBroadcast(Context context,
2                                    int requestCode,
3                                    Intent intent,
4                                    int flags) {
5   PendingIntent ret = PendingIntent.getBroadcast(
6      context, requestCode, intent, flags);
7   getInstance().intentMap.put(ret, intent);
8   return ret;
9 }
```

Figure 4.7: Recording of created pending intents.

map `intentMap` to record the relationship between `PendingIntent` and the `Intent` it warps. This map is latter used when scheduling the equivalent `JobService`.

Figure 4.7 shows the inserted static method. In its definition, it first calls the original `PendingIntent` allocation API (line 5). Before returning the allocated instance of `PendingIntent` (line 8), it uses `intentMap` to save the relationship between the `PendingIntent` and the wrapped `Intent` (line 7). The `intentMap` is a `WeakHashMap`, which will not prevent garbage collection of its keys and should not cause memory leaks.

4.4.2 Refactoring the Scheduling of Alarms

When generating an output quadruple (t, d, i, cl), the call site of this alarm scheduling is also included. We use this call site information to perform code refactoring, which convert the alarm scheduling into a `JobService` scheduling. There are several APIs that can perform alarm scheduling. Here we use the most common API `set` as example; its signature is `void set(int, long, PendingIntent)`.

```
1  void set(int type,
2        long triggerAtMillis,
3        PendingIntent operation) {
4    long exactDelay = extractExactDelay(type, triggerAtMillis);
5    Class<?> tgtJobService =
6        getClassObjFromStr(getJobServiceForPendingIntent(operation));
7    scheduleJobServiceOneShot(getInstance().context,
8        exactDelay, operation, tgtJobService);
9  }
10 static void scheduleJobServiceOneShot(Context context,
11                                       long startDelay,
12                                       PendingIntent pi,
13                                       Class<?> jobserviceClass) {
14   JobScheduler js = context.getSystemService(Context.JOB_SCHEDULER_SERVICE);
15   int jobID = pickJobID(js, pi);
16   JobInfo.Builder builder = new JobInfo.Builder(jobID,
17           new ComponentName(context, jobserviceClass));
18   builder.setMinimumLatency(startDelay);
19   PersistableBundle bundle = new PersistableBundle();
20   Intent i = getInstance().intentMap.get(pi);
21   packIntent(bundle, i);
22   builder.setExtras(bundle);
23   JobInfo info = builder.build();
24   js.schedule(info);
25 }
```

Figure 4.8: Example of inserted method.

The inserted `PrestoJobServiceFramework` class contains a method that has the same signature as `set`. This new method will perform the necessary preparation for `JobService` scheduling and will schedule the constructed `JobService`, as explained later. We replace calls to `set` with calls to this new method.

Figure 4.8 shows the simplified implementation of this method. The signature is the same as that of `set` (line 1). The method first converts the start delay used in alarm scheduling into the format used in `JobService` scheduling by calling helper method `extractExactDelay` (line 4). It then retrieves the `Class` object of the constructed `JobService` using helper method `getJobServiceForPendingIntent` and calls the helper method `scheduleJobServiceOneShot` to perform the actual `JobService` scheduling (lines 5–8).

Unlike the scheduling of an alarm, which puts time and target information in the parameter of the scheduling API, `JobService` requires a `JobInfo` object when using the scheduling API. This object records necessary information such as start delay and whether this is a recurring job. A `JobInfo` builder is created with the necessary target and start delay setup (line 16–18). Because the original `Intent` object is required by each alarm target, we retrieve the original `Intent` from the `intentMap` and pack it into the `JobInfo` using helper method `packIntent` (line 19–22). The packing and unpacking is achieved by using APIs in Android's `Parcelable` interface, which are implemented by `Intent` objects. The `JobInfo` is then built by the builder and scheduled through the `schedule` API (line 24). In this manner, an inefficient alarm is converted into a `JobService`. The next subsection describes how to construct an equivalent `JobService` for the target of the original alarm.

4.4.3 Construction of JobService for an Alarm

In order to convert an energy-inefficient alarm, a `JobService` is constructed for each cl for a quadruple (t, d, i, cl) in the static analyses output (Section 4.3). Since the target classes of reported alarms are all subclasses of `BroadcastReceiver`, which has only one callback method `onReceive`, one way to perform the `JobService` construction is to create a blank subclass of `JobService` and invoke the `onReceived` method of the targeted `BroadcastReceiver` directly. This behavior ensure that the `BroadcastReceiver` will still have the same behavior as it did before the code refactoring.

Figure 4.9 shows a simplified `JobService` class constructed for the running example from Figure 4.1. When the system starts the `JobService` `SyncAlarmReceiverRef`,

```
1  class SyncAlarmReceiverRef extends JobService {
2    boolean onStartJob(JobParameters jobParameters) {
3      PersistableBundle bundle = jobParameters.getExtras();
4      Intent i = PrestoJobServiceFramework.unpackIntent(bundle);
5      BroadcastReceiver target = new SyncAlarmReceiver();
6      target.onReceive(this.getApplicationContext(), i);
7      return false;
8    }
9    boolean onStopJob(JobParameters jobParameters) {
10     return false;
11   }
12 }
```

Figure 4.9: Example of constructed `JobService` for Moloko app.

the `onStartJob` callback method will be called. The packed `Intent` object is retrieved from parameter `jobParameters`, which wraps the `JobInfo` object created at `JobService` scheduling (lines 3–4). An instance of original the `SyncAlarmReceiver` class will be created and its `onReceive` callback method will be invoked with the current application context and the unpacked `Intent` object. The `onStartJob` callback returns false to indicate that this `JobService` does not need a system-managed wake lock to run in the background.

After these steps, the application will have it energy-inefficient alarms converted to `JobServices`. In the next section, we demonstrate the reduction of energy consumption caused by these refactoring steps.

4.5 Evaluation

We implemented the static detector of inefficient `AlarmManager` usage and automated static code refactoring component using the Soot analysis framework [77].

We crawled the entire F-Droid [21] repository and downloaded the top 100 apps from each category from Google Play Store [29]. The static detection was performed on the entire subset of these apps that have uses of `AlarmManager`, 41 apps were reported to satisfy the definition of inefficient `AlarmManager` usage defined in Section 4.1.5. When testing these apps on a real device, there were 3 apps that failed to start or required a paid subscription. These apps were removed from our test sets. The final set of refactored apps contained the remaining 38 apps.

The evaluation was conducted on a Google Nexus 5X device with Android 7.1.2 installed. The device was factory reset before the evaluation. We first installed all original 38 apps to the device and performed necessary initial setups to allow applications to finish scheduling their alarms for background tasks. The device was then charged to 100% battery and connected to Wi-Fi and placed still on a table for 12 hours. Due to the fact that the battery level can only be measured using Android APIs when device is woke up, we instrumented each app to send a log with the current battery level to the Android log system when it wakes up the device due to an alarm. This approach is preferable to measuring the battery level at a fixed interval, which would inevitably cause more device wake ups, causing imprecision in battery level measurements. These logs were then collected and analyzed. After these steps were completed, we performed a factory reset to remove all installed apps and then installed the refactored 38 apps. We performed the same setup steps, connected the device to the same Wi-Fi, charged the device to 100% battery, and placed at same location for another 12 hours. These apps also had the same instrumentation for logging purposes.

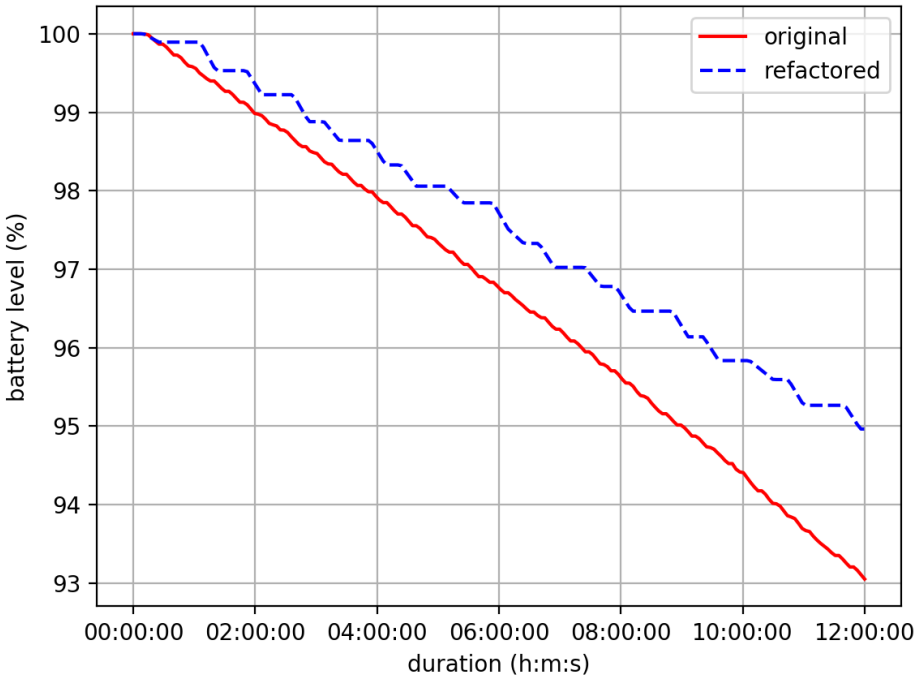

Figure 4.10: Battery level comparison

Figure 4.10 shows the battery level comparison with the two sets of apps installed on the device, over a 12-hour period. The solid line represents the changes of battery level when device was installed with original test apps. The dotted line represents the changes of battery level when device was installed with the refactored apps. As shown in the figure, the smoothness of the two lines is different. This is caused by the different device wake up intervals for the two sets of apps. The original apps wake up the device every 3–7 minutes while for the refactored apps this number is 10-20 minutes. In this case, there are fewer data points in the logs generated with refactored apps, compared to the ones generated with original apps. The figure also shows that after the refactoring process, the battery drop is reduced. At the end of

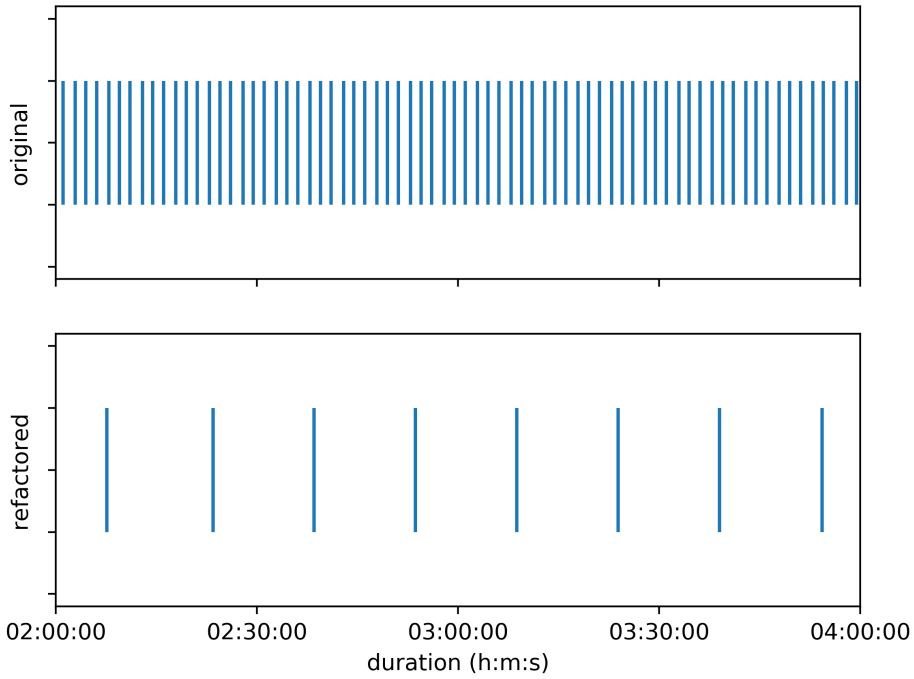

Figure 4.11: Device wake up comparison

the 12 hours testing period, the original apps consumed nearly 7% of battery capacity while the refactored apps consumed 5%. We also calculated the average discharge rate of 12-hour test for these two sets of apps. The average discharge rate of the set of original apps is $15.50mA$ while the average discharge rate of the set of refactored apps is $10.95mA$.

Figure 4.11 shows the device wake ups for the period of 02:00:00 to 04:00:00 of the 12 hour test period. Each vertical line in the figure represents a device wake up. As it can be concluded from the figure, the wake ups are much more frequent for the set of original apps than for the set of refactored apps. With the original apps, roughly every 5 minutes there is a device wake up. For the modified apps, this

number is around 15 minutes. In this figure, we did not show the entire 12 hour testing period as the subfigure for the original apps would be too dense. For the entire 12 hours, there are 919 device wake ups caused by the set of original apps. For the set of refactored apps, this number is 100. The reduced device wake ups explains the reduced battery consumption of the refactored apps.

4.6 Conclusions

This work shows that energy inefficiency in `AlarmManager` usage can be detected using static analysis and can be eliminated by automated code refactoring. The refactored apps have the same functionality but cause significantly fewer device wake ups when they are installed and running together. As a result, battery drain is slowed down. This work illustrates how static analysis and code rewriting can be beneficial when evolving Android apps to reflect the evolution of the underlying Android framework.

CHAPTER 5: Related Work

5.1 Energy Analysis for Android

The problem of optimizing the energy consumption of mobile devices has been the subject of various studies over the last few years. Some optimization techniques consider the screen's energy consumption and reduce it by modifying the app's color scheme [47, 53]. The network usage is another significant consumer of energy and some profiling techniques have been proposed to characterize this usage [71]. Another source of energy inefficiencies is the misuse of background services and Android APIs with energy impact [15, 52].

There exists work on energy profiling in order to discover energy inefficient Android apps. Xinbo and Ziliang [88] developed a software-based profiler to measure the energy consumption of different programming language and runtime system in Android. Their results show that the latest Android ART runtime has less energy consumption than traditional Android dalvik runtime. Nucci et al. [64] developed a software-based energy profiler for Android apps which can achieve accurate results when compared to a hardware energy monitor. Other work in this area [34, 46, 67, 68] uses dynamic analysis to characterize the energy consumption on a given app. While these profilers can help debug energy-inefficient apps, they cannot directly identify the underlying problems based only on the profiling results, since an increase in energy consumption does not necessarily imply the presence of an energy bug or inefficiency.

Some researchers have used crowdsourcing techniques to obtain data from a large number of devices, which allows more reliable identification of apps that cause battery drain [60, 66]. As with any profiling techniques, there is a significant challenge in obtaining representative samples of run-time executions on real devices. Triggering such executions by the developers is further complicated by the event-driven nature of Android apps. One advantage of static detection is that it allows for problems to be detected early and without run-time information.

Leak Detection There are various techniques for detecting leaks in Java programs. Torlak and Chandra [80] designed a static interprocedural analysis to discover resources leaks (e.g., for I/O streams). Kondoh and Onodera [42] performed static analysis on Java programs with JNI to uncover leaks in error handling code. Static detection of memory leaks for Java has also been considered (e.g., [90]).

GreenDroid [55] uses Java PathFinder to traverse UI event handler callbacks, based on a UI model constructed manually, in order to uncover resource leaks dynamically. A similar approach by Banerjee et al. [10] uses a modified version of Dynodroid [58] to perform dynamic analysis of resource leaks. Their later work [11] uses an instrumentation utility to instrument potential defective Android applications to collect runtime method traces to pin-point energy related leaks. Zhang et al. [98] used dynamic taint analysis to detect design flaws related to network operations. However, like any dynamic taint analysis approach, this approach introduces relatively high run-time overhead. Jun et al. [41] developed a dynamic leak detector for Android activities and fragments, based on UI traversal and memory profiling; this technique was not used to analyze energy-related leaks. One limitation of these

techniques is the difficulty of achieving high code coverage in run-time execution of Android apps [18]. Furthermore, some techniques [10] require physical measurement of voltage and current of the device, which is not easy to achieve in practice. Static analysis provides a viable alternative/complement to such dynamic approaches.

Researchers have also developed static leak detectors for Android apps. Guo et al. [33] use static analysis to find resource-related API invocations in event handler callbacks and to reason about possible resource leaks. As the authors of that prior work state, the limitations of their control-flow analysis lead to false negatives. A somewhat similar approach, with similar limitations, was proposed by Pathak et al. [69], with the additional problem that user-provided control-flow ordering of event handlers is needed. Wu et al. [87] developed a tool which can perform interprocedural analysis on Android applications to statically identify potential leaks. The analysis makes a variety of simplifying assumptions about the data flow and the control flow of apps, and does not appear to use any systematic static model of callback ordering or leak pattern definitions. Similarly, Jiang et al. developed a static analysis tool [40] which can generate callback sequences of Android application in order to identify unreleased resources. This approach also makes simplifying assumptions about the control flow. A static analysis on Android wake lock misuses was proposed by Liu et al. [56], however, the generation of applications' method call sequences is relatively simplified which may miss some feasible sequences and may cause false negatives.

5.2 Test Generation for Android

There are various techniques for automated testing for mobile apps. Choudhary et al. [18] present a summary of existing testing approaches for Android. The Monkey testing tool [61], provided by Google, generates UI events randomly. Android GUI Ripper [3] and MobiGUITar [4] create a dynamically built GUI model and test cases based on it. Yang et al. [95] also explore the application dynamically, but use static analysis to determine relevant UI events for a specific activity. CrashScope [62] uses a similar model-based approach; however, it focuses on detecting and reporting application run-time crashes. The A^3E GUI exploration tool [8] employs both dynamic depth-first exploration, similarly to Android Ripper, as well as targeted exploration based on a model derived from static analysis. Work by Zhang et al. [96, 97] generates UI tests based on the WTG model in order to expose resource leaks, but does not analyze acquire/release sequences for these resources. Jensen et al. [39] generate event sequences using a UI model and event handler summaries derived from static analysis. ACTEve [5] performs concolic testing by symbolically tracking UI events from their origin to their handler. There are also examples of using machine learning techniques to improve automated test generations. SwiftHand [17] achieves code coverage by learning and exploring an abstraction of the app's GUI. Grano et al. [31] designed a automated test generation tool based on machine learning from users' reviews. Other representative tools include EvoDroid [59], Droidmate [38], Dynodroid [58] and PUMA [35].

5.3 Automated Refactoring for Android

There is a growing interest in using automatic code refactoring to improve Android applications. Refactoring tools [49, 51] help developer to encapsulate consuming tasks into `AsyncTask` and `IntentService` which help ease the application development. Other work focused on energy optimization. Optimizers [14, 36, 45] show that energy consumption caused by tail-energy of Android network access can be reduced by batching network requests using automatically code refactoring. Banerjee and Roychoudhury used automatically constructed event flow graph to optimize energy related resource usage [9, 12]. It shows a significant improvement in energy efficiency. Linares-Vásquez et al. introduced GEMMA [54], which automatically change the color scheme of Android apps to reduce their energy consumption on devices with (AM)OLED displays. A similar approach is used by Agolli et al., however, their work [1] is focused on making the color change indistinguishable while reducing energy consumption. Li and Halfond introduced Leafactor[19], which perform refactor on Android layout files to improve energy efficiency of UI rendering. Zhang et al. proposed a way [100] to offload computing intensive tasks to cloud by refactoring Android application's source code. The results show reduced of execution time and energy consumption.

5.4 Static Control-Flow Analysis for Android

There is a significant amount of work on security analysis [16, 22, 23, 30, 37, 48, 57, 65, 101] performs static modeling of certain aspects of Android control flow (and the related data flow). However, none of these approaches provides a comprehensive model of GUI control flow or provides possible sequences of callbacks. Work [8] uses

static analysis to construct an activity transition graph and use it to perform run-time GUI exploration. However, it does not capture the GUI effects of callbacks and window stack changes. There are other static analyses perform modeling of callback sequences in Android, for example, for the purposes of race detection (e.g., [50]) and static checking (e.g., [70, 99]); all these approaches employ ad hoc control-flow modeling that lacks generality. FlowDroid [7] uses another approach to model the event-driven control flow. This work creates an artificial main method to represent the invocation of callbacks from Android framework to the application code. While this approach partially models the possible sequences of event handlers, it does not account for the full generality of GUI effects of event handlers, and does not represent precisely the interleaving of callbacks that span multiple activities and their lifetimes. Related work [91–94] provides a more comprehensive solution which considers the relevant control-flow information in the window transition graph described earlier. The modeling of window stack is also defined in this work. In addition to serving as the basis of the resource leaks detection described in this work, the modeling of Window Transition Graph can be potentially be useful for other purpose, as illustrated in [85, 86, 97].

CHAPTER 6: Conclusions

The Android platform has experienced rapid development over the recent years. The event-based control flow and limited hardware resources bring new issues in software development as well as new challenges in program analyses. Due to the limited resource on Android devices, energy-related defects are of significant importance. Thus, we focused on detecting and eliminating energy-inefficiency patterns in Android applications.

We conducted case studies on real-world energy defects and confirmed observations by other researchers that one common pattern of energy defects comes from "missing deactivation" of energy-related resources. In Chapter 2, we focus on missing deactivation of Android GPS resources as it can cause considerable battery drain. We first define two patterns of GPS resource leaking behaviors. The definition is based on formal definitions of relevant aspects of Android GUI run-time control flow, including modeling of GUI events, event handlers, transitions between windows, and the associated sequences of callbacks. We then propose a static analysis for detection of energy-related defects due to Android location listener leaking. The technical foundation for this analysis is the static modeling of possible sequences of window transitions and their related callbacks. We perform control-flow analysis of individual callbacks and combined it with analysis of callback sequences to identify instances of two location listener leaking patterns. We evaluate the analysis on seventeen known

and new defects that were detected in previously-analyzed and never-analyzed applications. All but one of the reported defects are observable at run time. The evaluation shows this approach is effective and the cost of the analysis is low.

In Chapter 3, we switch our focus to sensor resource leak defects. Similarly to the Android GPS leak defects discussed in Chapter 2, a missing deactivation of Android sensor listeners can cause energy draining issues. It is also clearly stated in the Android developer guidelines that sensor resources should be properly released when an application is going to idle state. We proposed a static analysis to detect potential sensor leaks in Android apps using context-free-language reachability analysis with automated test generation to perform verification. The analysis is performed on a static graph model of sensor-related objects and API calls. Each node represents a Android window and each edge is labeled by symbols representing the opening/closing of UI windows and acquiring/releasing of sensors. We defined two context-free languages over the alphabet of symbols. Each graph path that defines a string from these languages represents a potential leak. The static analysis identifies and report such paths, which are then used to generate test cases to verify the leaks on a real device. Experimental studies indicate that analysis precision is high and that sensor leaks in realistic Android apps can be successfully detected.

There are other types of energy-inefficiency patterns that are unrelated to leaks. One aspect is inefficient usage of energy-intensive system services. In Chapter 4, we focus on energy-inefficiency usage of `AlarmManager`. Due to legacy reasons, there are still applications that use alarms scheduled by `AlarmManager` to perform periodic background tasks. This causes frequent device wake ups and increases battery

consumption. We proposed a static analysis to detect such inefficient usages and designed an automated code refactoring engine to convert such usages to Android jobs managed by `JobScheduler`. We conducted tests on a real device and showed that this approach reduced the device wake ups as well as the battery consumption.

In summary, the results from our work shows that static analysis is effective and precise for detection of energy-inefficiency patterns in Android applications. Such analysis requires no manual efforts and is efficient when performed at a large scale, suggesting that it is suitable for practical use in static checking tools for Android.

BIBLIOGRAPHY

[1] Tedis Agolli, Lori Pollock, and James Clause. Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 30–34, 2017.

[2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *International Conference on Automated Software Engineering*, pages 258–261, 2012.

[4] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, pages 53–59, 2015.

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[6] Android Development Documentation Intent. 2017. developer.android.com/reference/android/content/Intent.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[8] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 641–660, 2013.

[9] Abhijeet Banerjee and Abhik Roychoudhury. Automated re-factoring of Android Apps to enhance energy-efficiency. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 139–150, 2016.

[10] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 588–598, 2014.

[11] Abhijeet Banerjee, Haifeng Guo, and Abhik Roychoudhury. Debugging energy-efficiency related field failures in mobile apps. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 127–138, 2016.

[12] Abhijeet Banerjee, Lee Kee Chong, Clement Ballabriga, and Abhik Roychoudhury. Energypatch: Repairing resource leaks to improve energy-efficiency of Android apps. In *IEEE Transactions on Software Engineering*, pages 470–490, 2017.

[13] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, pages 27–38, 2012.

[14] Huaqian Cai, Ying Zhang, Zhi Jin, Xuanzhe Liu, and Gang Huang. Delaydroid: Reducing tail-time energy by refactoring android apps. In *Internetware*, pages 1–10, 2015.

[15] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *International Conference on Mobile Computing and Networking*, pages 40–52, 2015.

[16] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011.

[17] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of Android apps with minimal restart and approximate learning. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 623–640, 2013.

[18] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for Android: Are we there yet? In *International Conference on Automated Software Engineering*, pages 429–440, 2015.

[19] Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 205–206, 2017.

[20] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[21] F-Droid Android App Repository. 2017. f-droid.org.

[22] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 576–587, 2014.

[23] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, College Park, 2009.

[24] GATOR: Program Analysis Toolkit For Android. 2017. web.cse.ohio-state.edu/presto/software/gator.

[25] Google. Tasks and back stack. developer.android.com/guide/components/tasks-and-back-stack.html, 2015.

[26] Google. Managing the activity lifecycle, 2015. developer.android.com/training/basics/activity-lifecycle.

[27] Google. Android dialogs. developer.android.com/guide/topics/ui/dialogs.html, 2015.

[28] Google. Location strategies. developer.android.com/guide/topics/location/strategies.html, 2015.

[29] Google Play app store. play.google.com/store/apps.

[30] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Network and Distributed System Security Symposium*, 2012.

[31] Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, and Harald Gall. Exploring the integration of user feedback in automated testing of Android applications. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 72–83, 2018.

[32] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23 (6):685–746, 2001.

[33] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in Android applications. In *International Conference on Automated Software Engineering*, pages 389–398, 2013.

[34] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *International Conference on Software Engineering*, pages 92–101, 2013.

[35] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *International Conference on Mobile Systems, Applications, and Services*, pages 204–217, 2014.

[36] Gang Huang, Huaqian Cai, Maciej Swiech, Ying Zhang, Xuanzhe Liu, and Peter Dinda. Delaydroid: an instrumented approach to reducing tail-time energy of android apps. *Science China Information Sciences*, 60(1):012106, Nov 2016. ISSN 1869-1919. doi: 10.1007/s11432-015-1026-y. URL https://doi.org/10.1007/s11432-015-1026-y.

[37] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *International Conference on Software Engineering*, pages 1036–1046, 2014.

[38] K. Jamrozik, P.V. Styp-Rekowsky, and A. Zeller. BOXMATE, 2017. boxmate.org.

[39] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 67–77, 2013.

[40] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. Detecting energy bugs in Android apps using static analysis. In *International Conference on Formal Engineering Methods*, pages 192–208, 2017.

[41] Ma Jun, Liu Sheng, Yue Shengtao, Tao Xianping, and Lu Jian. LeakDAF: An automated tool for detecting leaked activities and fragments of Android applications. In *IEEE Annual Computer Software and Applications Conference*, pages 23–32, 2017.

[42] Goh Kondoh and Tamiya Onodera. Finding bugs in Java native interface programs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 109–118, 2008.

[43] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, 2003.

[44] Ondrej Lhoták. Spark: A scalable points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[45] Ding Li and William G. J. Halfond. Optimizing energy of http requests in android applications. In *DeMobile*, pages 25–28, 2015.

[46] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for Android applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 78–89, 2013.

[47] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. Nyx: A display energy optimizer for mobile web apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 958–961, 2015.

[48] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 21–32, 2013.

[49] Yu Lin and Danny Dig. Refactorings for android asynchronous programming. In *International Conference on Automated Software Engineering*, pages 836–841, 2015.

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

[51] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming. In *International Conference on Automated Software Engineering*, pages 224–235, 2015.

[52] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *International Conference on Mining Software Repositories*, pages 2–11, 2014.

[53] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Optimizing energy consumption of GUIs in Android apps: A multi-objective approach. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 143–154, 2015.

[54] Mario Linares-Vásquez, Carlos Cárdenas, and Gabriele Bavota. Gemma: Multi-objective optimization of energy consumption of GUIs in Android apps. In *International Conference on Software Engineering Companion*, pages 11–14, 2017.

[55] Yepang Liu, Chang Xu, S. C. Cheung, and Jian Lu. GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, 40:911–940, September 2014.

[56] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. Understanding and detecting wake lock misuses for android applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 296–409, 2016.

[57] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *ACM Conference on Computer and Communications Security*, pages 229–240, 2012.

[58] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *ACM SIGSOFT International Symposium*

on the Foundations of Software Engineering, pages 224–234, 2013.

[59] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 599–609, 2014.

[60] Chulhong Min, Youngki Lee, Chungkuk Yoo, Seungwoo Kang, Sangwon Choi, Pillsoon Park, Inseok Hwang, Younghyun Ju, Seungpyo Choi, and Junehwa Song. PowerForecaster: Predicting smartphone power impact of continuous sensing applications at pre-installation time. In *ACM Conference on Embedded Networked Sensor Systems*, pages 31–44, 2015.

[61] Monkey UI/Application Exerciser for Android. developer.android.com/tools/help/monkey.html.

[62] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *IEEE International Conference on Software Testing, Verification, and Validation*, pages 1–12, 2018.

[63] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[64] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of Android apps: Simple, efficient and reliable? In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 103–114, 2017.

[65] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves le Traon. Effective inter-component communication mapping in Android with Epicc. In *USENIX Security*, pages 543–558, 2013.

[66] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *ACM Conference on Embedded Networked Sensor Systems*, pages 10:1–10:14, 2013.

[67] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *European Conference on Computer Systems*, pages 153–168, 2011.

[68] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? In *European Conference on Computer Systems*, pages 29–42, 2012.

[69] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *International Conference on Mobile Systems, Applications, and Services*, pages 267–280, 2012.

[70] Etienne Payet and Fausto Spoto. Static analysis of Android programs. *IST*, 54 (11):1192–1201, 2012.

[71] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Profiling resource usage for mobile applications: A

cross-layer approach. In *International Conference on Mobile Systems, Applications, and Services*, pages 321–334, 2011.

[72] RememberTheMilk: Remember the milk website. 2017. www.rememberthemilk.com/.

[73] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[74] Atanas Rountev and Dacong Yan. Static reference analysis for GUI objects in Android software. In *International Symposium on Code Generation and Optimization*, pages 143–153, 2014.

[75] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, pages 126–137, 2003.

[76] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[77] Soot Analysis Framework. 2017. www.sable.mcgill.ca/soot.

[78] Manu Sridharan and Rastislav Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[79] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–293, 2000.

[80] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *International Conference on Software Engineering*, pages 535–544, 2010.

[81] UI Automator Testing Framework. 2017. developer.android.com/training/testing/ui-automator.html.

[82] UI Automator Wrapper. Python wrapper of Android UI Automator test tool, 2017. github.com/xiaocong/uiautomator.

[83] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, pages 18–34, 2000.

[84] WakefulBroadcastReceiver for Android. developer.android.com/reference/android/support/v4/content/WakefulBroadcastReceiver.

[85] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in Android applications. In *International Conference on Compiler Construction*, pages 185–195, 2016.

[86] Haowei Wu, Yan Wang, and Atanas Rountev. Sentinel: Generating GUI tests for Android sensor leaks. In *IEEE/ACM International Workshop on Automation of Software Test*, pages 27–33, 2018.

[87] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. Relda2: An effective static analysis tool for resource leak detection in Android apps. In

International Conference on Automated Software Engineering, pages 762–767, 2016.

[88] Chen Xinbo and Zong Ziliang. Android app energy efficiency: The impact of language, runtime, compiler, and implementation. In *IEEE International Conference on Big Data and Cloud Computing*, pages 485–492, 2016.

[89] Dacong Yan. *Program Analyses for Understanding the Behavior and Performance of Traditional and Mobile Object-Oriented Software*. PhD thesis, Ohio State University, July 2014.

[90] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. LeakChecker: Practical static memory leak detection for managed languages. In *International Symposium on Code Generation and Optimization*, pages 87–97, 2014.

[91] Shengqian Yang. *Static Analyses of GUI Behavior in Android Applications*. PhD thesis, Ohio State University, September 2015.

[92] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *International Conference on Software Engineering*, pages 89–99, 2015.

[93] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static window transition graphs for Android. In *International Conference on Automated Software Engineering*, pages 658–668, 2015.

[94] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for

Android. *International Journal of Automated Software Engineering*, pages 1–41, June 2018.

[95] Wei Yang, Mukul Prasad, and Tao Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265, 2013.

[96] Hailong Zhang and Atanas Rountev. Analysis and testing of notifications in Android Wear applications. In *International Conference on Software Engineering*, pages 64–70, 2016.

[97] Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated test generation for detection of leaks in Android applications. In *AST*, pages 64–70, 2016.

[98] Lide Zhang, Mark S. Gordon, Robert P. Dick, Z. Morley Mao, Peter Dinda, and Lei Yang. ADEL: An automatic detector of energy leaks for smartphone applications. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 363–372, 2012.

[99] Sai Zhang, Hao Lü, and Michael D. Ernst. Finding errors in multithreaded GUI applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 243–253, 2012.

[100] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. Refactoring android java code for on-demand computation offloading. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 233–248, 2012.

[101] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104, 2012.