Automated Analysis of Energy Efficiency and Execution Performance for Mobile Applications

Yepang Liu

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

June 2015, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

_____

Yepang Liu

22 June, 2015

Automated Analysis of Energy Efficiency and Execution Performance for Mobile Applications

by

Yepang Liu

This is to certify that I have examined the above PhD thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Prof. Shing-Chi Cheung, Thesis Supervisor

Prof. Qiang Yang, Head of Department

Department of Computer Science and Engineering

22 June, 2015

To the memory of my grandfather

and

To my family

Acknowledgments

I would like to thank numerous people who have helped and supported me throughout my graduate studies.[1]

First and foremost, I would like to express my most sincere appreciation and thanks to my advisor Prof. Shing-Chi Cheung, who has been a tremendous mentor during my five years of study at HKUST. Prof. Cheung contributed so much to my research by continually inspiring me to find interesting problems, guiding me to solve problems elegantly, and training me to write high-quality technical papers. Without his persistent support and invaluable advices, this thesis would not have been possible. I am also grateful for the excellent example Prof. Cheung has set to us students as a kind, patient, generous, hard-working and responsive person.

Secondly, I would also like to express my gratitude to Dr. Chang Xu, who has also been an important mentor to me. In fact, I came up with my very first research idea during an insightful discussion with him shortly after I joined HKUST. Ever since then, he has been very supportive for my research. I will always remember that he flied to Hong Kong and worked side by side with me on my first paper. Even though publishing this paper was the most difficult time during my graduate studies, I did not lose faith in myself and I believe it was his encouragements and positive attitude that gave me strength. I am fortunate to have such a friend and collaborator.

Thirdly, I want to sincerely thank my thesis proposal and examination committee: Prof. Li Qiu, Prof. Mauro Pezzè, Prof. Chi-Ying Tsui, Prof. Charles Zhang, Prof. Sung Kim, Prof. Raymond Wong, and Prof. Jogesh Muppala for spending time reviewing my thesis draft. Their valuable comments guided me to further im-

[1]The research conducted in this thesis was supported by Research Grants Council (General Research Fund 611912 and 611813).

prove the quality of my thesis and inspired me to look at the research problem from different perspectives.

Fourthly, I am also grateful to many other people at HKUST. I would like to thank my dear friends here for bringing so much fun and happiness to my life. I would also like to thank my group mates for creating a supportive yet competitive research environment. My thanks also extends to those students, especially Sathish Raghuraman, who helped me collect research data, conduct experiments and cross-validate results.

Finally, my deepest gratitude goes to my family. Words cannot express how grateful I am to my beloved wife, parents and grandparents. Without their love and support, I would not have been come thus far on the academic path.

Table of Contents

List of Figures

List of Tables

Automated Analysis of Energy Efficiency and Execution Performance for Mobile Applications

by Yepang Liu

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

Abstract

Mobile applications' energy efficiency and performance have a vital impact on user experience. However, many mobile applications on market suffer from bugs that can cause significant energy waste and performance degradation, thereby losing their competitive edge. Locating these bugs is labor-intensive and thus automated diagnosis is highly desirable. Unfortunately, people have limited understanding of these bugs and there are no clear criteria to facilitate automated analysis of mobile applications' energy efficiency or execution performance. To bridge the gap, we conducted two large-scale empirical studies of real-world energy and performance bugs from popular Android applications. We studied the characteristics of these bugs and identified several common causes of energy waste and performance degradation.

For energy bugs, we observed that (1) forgetting to deactivate device sensors or wake locks after use and (2) ineffectively utilizing sensory data can cause serious energy waste. To help developers detect such energy bugs, we proposed a dynamic analysis technique GreenDroid. GreenDroid automatically generates user interaction event sequences to systematically execute an Android application for state space exploration. During execution, it tracks the transformation, propagation and consumption of sensory data and analyzes whether the data are effectively utilized by

the application to bring users perceptible benefits. It also closely monitors whether device sensors and wake locks are properly deactivated after use. We evaluated GreenDroid using 14 popular open-source Android applications. GreenDroid successfully located 13 real energy bugs in these applications and additionally found two previously-unknown bugs that were later confirmed by developers.

For performance bugs, we observed that (1) conducting lengthy operations in an application's main thread and (2) frequently invoking heavy-weight program callbacks can seriously reduce the responsiveness of an application. To help developers detect such performance bugs, we designed a light-weight static analysis technique PerfChecker. PerfChecker automatically scans an Android application's bytecode and identifies a set of checkpoints whose efficiency is critical. It then analyzes whether the checkpoints' implementation satisfies the efficiency rules formulated from our empirical study. We evaluated PerfChecker with 39 popular and large-scale Android applications (29 open-source and 10 commercial) and a widely-used library. PerfChecker successfully detected 178 previously-unknown performance bugs, among which 88 were quickly confirmed by developers and 20 critical ones were fixed soon afterwards. We also confirmed via comparison experiments that fixing our detected performance bugs can significantly improve the performance of the corresponding applications.

Chapter1

Introduction

Mobile devices such as smartphones and tablets have become the de facto computing platforms in our daily lives [84]. One major reason behind their incredible market success is that applications running on these devices can assist their users in a wide variety of daily activities (e.g., work and entertainment). Take one of the most popular mobile computing platforms Android for an example. Up until July 2014, there were already 1.3 million applications available on its official Google Play store [19]. These applications span across 41 different categories and had received more than 50 billion downloads from users around the world [20].

Since users rely on various mobile applications during their daily lives, the user experience of such applications becomes vital. Undoubtedly, in the mobile computing era, application functionality still remains as a key factor that affects user experience. Nonetheless, with more and more applications of similar functionalities emerge on market (e.g., various web browsers), non-functional properties such as energy efficiency and execution performance have also gradually become important factors.

However, our inspection of 60,000 Android applications randomly sampled from Google Play store revealed an alarming fact: 11,108 (18.5%) of them have suffered or are suffering from energy and performance bugs [67]. *Energy bugs* can silently and quickly drain the mobile devices' battery power. *Performance bugs* can significantly slow down mobile applications and cause them to consume an excessive amount of computational resources (e.g., memory and network bandwidth). These bugs had a severe impact on user experience and caused significant user frustrations [79].

The pervasiveness of energy and performance bugs in mobile applications is attributable to two major reasons. First, mobile devices are usually resource-

constrained (e.g., limited battery power and memory), but the applications running on them often have to conduct energy-consumptive and computationally-intensive tasks such as network communication and graphics rendering. Small inefficiency in the applications' implementation may lead to noticeable energy waste and performance degradation. Second, many mobile applications are developed by individual developers without dedicated quality assurance. It is hard for them to exercise due diligence in assuring application energy efficiency and performance, especially when they have to push their application products to market in a short time due to the fierce competition.

Locating energy and performance bugs in mobile applications is the first step towards energy consumption and performance optimization. However, it is a difficult task for developers because energy inefficiency and performance degradation often only occur at certain application states. To manifest the bugs, developers often have to extensively test their applications on different devices and perform energy and performance profiling (e.g., measuring energy or memory consumption). To figure out the root causes, they often have to carefully instrument the concerned programs to collect a large amount of runtime information for offline investigation. Such a process is tedious and labor-intensive. Therefore, automated diagnosis techniques are highly desirable. This motivates our research.

The goal of our research is to design useful and automated analysis techniques to help developers quickly locate energy and performance bugs in their mobile applications. We restrict our study scope to Android smartphone applications due to their platform openness and popularity. To achieve the goal of our research, we need to address two important problems:

- **Energy and performance bug understanding.** First of all, mobile computing platforms are relatively new. Both research communities and industries lack a good understanding of the energy and performance bugs in the applications running on these platforms. However, in order to design useful techniques to help developers combat such bugs, we must obtain a deep understanding of these bugs in the first place.

- **Automated oracles for judging energy and performance bugs.** Second, energy bugs may silently drain battery power and performance bugs may gradually lead to performance degradation. Such bugs rarely cause immediate fail-stop consequences (e.g., crash). This makes it difficult to judge their existence. However, to design automated analysis techniques, we need decidable criteria to facilitate mechanical judgment of energy inefficiency and performance degradation.

To address these problems, we conducted two large-scale empirical studies of real-world energy and performance bugs collected from popular Android applications. We carefully studied the characteristics of these bugs such as how they manifest themselves and the difficulties in their diagnosis, and identified several common causes of energy waste and performance degradation.

In the empirical study of energy bugs, we observed two common causes of energy waste:

- **Missing sensor or wake lock deactivation.** Many Android applications use smartphones' built-in sensors (e.g., GPS) to continuously probe users' physical and cyber environments to provide context-aware services (e.g., navigation). To use a sensor, an application needs to register a listener with the Android OS. The listener should be unregistered when the sensor is no longer being used [1]. Similarly, to make a phone stay awake for computation, an application has to acquire a wake lock from the Android OS. The acquired wake lock should also be released as soon as the computation completes. Forgetting to unregister sensor listeners or release wake locks could quickly deplete a fully charged phone battery.

- **Sensory data underutilization.** Sensing operations consume considerable energy, and therefore the obtained sensory data should be effectively utilized by the applications to bring users perceptible benefits. Poor sensory data utilization (e.g., being used to render invisible GUIs) often results in energy waste.

To automatically detect such energy bugs, we proposed a dynamic analysis technique GreenDroid. GreenDroid generates user interaction event sequences to systematically execute an Android application for state space exploration. During execution, it tracks the transformation, propagation and consumption of sensory data and analyzes whether the data are effectively utilized by the application at each explored state. It also monitors whether sensors/wake locks are properly registered/acquired and unregistered/released. To evaluate the usefulness of GreenDroid, we applied it to analyze 14 popular open-source Android applications. GreenDroid completed energy efficiency analysis for these applications in a few minutes. It successfully located 13 real energy bugs in these applications and additionally found two previously unknown bugs, which were later confirmed by developers. We were also invited by developers to make patches for the two new bugs and one of our patches was accepted. These evaluation results confirm GreenDroid's efficiency and effectiveness.

In the empirical study of performance bugs, we also observed two common causes of performance degradation:

- **Lengthy operations in main threads.** Android applications by default run entirely on a single thread "main thread". This thread handles user interaction events and therefore any methods running in the thread should do as little work as possible to keep the application responsive. Running lengthy operations in the main threads can significantly reduce the applications' responsiveness.

- **Frequently invoked heavy-weight callbacks.** Android applications are event-driven and consist of a set of callbacks. Some callbacks are frequently invoked by Android OS and therefore need to be highly-efficient. Heavy-weight frequently-invoked callbacks can significantly slow down an application.

To automatically detect such performance bugs, we designed a light-weight static analysis technique PerfChecker. PerfChecker automatically scans an Android application's bytecode and identifies a set of checkpoints whose efficiency is critical. It then analyzes whether the checkpoints' implementation violates the efficiency rules formulated from studying the fixing patches of real-world performance bugs. To

evaluate the usefulness of PerfChecker, we conducted experiments on 39 popular and large-scale Android applications (29 open-source and 10 commercial) and a widely-used application development library. PerfChecker quickly finished analyzing each subject in a few seconds to a few minutes. It successfully detected 178 previously-unknown performance bugs, among which 88 were quickly confirmed by developers and 20 critical ones were fixed soon afterwards. We also confirmed via comparison experiments that fixing the detected performance bugs can indeed significantly improve the performance of our studied applications. These evaluation results confirm PerfChecker's efficiency and effectiveness.

To summarize, we make the following contributions in this thesis:[1]

- We conducted two large-scale empirical studies of real-world energy and performance bugs in Android applications. Our findings can help understand the characteristics of these bugs and provide guidance to related research.
- We proposed a dynamic analysis technique GreenDroid to help developers automatically diagnose common patterns of energy bugs in Android applications.
- We proposed a static analysis technique PerfChecker to help developers automatically detect common patterns of performance bugs in Android applications.
- We implemented GreenDroid and PerfChecker and evaluated them by extensive experiments on a large number of real-world popular Android applications. Our evaluation results confirmed the effectiveness and usefulness of the techniques.

Organization. The rest of this thesis is organized as follows. Chapter 2 introduces preliminary knowledge about Android platform and applications. Chapter 3 presents our empirical study of energy bugs in Android applications and our dynamic analysis technique for automatically diagnosing common patterns of energy bugs.

[1]We make our empirical study data and tool prototypes publicly available at these two websites for research purposes: `http://sccpu2.cse.ust.hk/greendroid/` and `http://sccpu2.cse.ust.hk/perfchecker`.

Chapter 4 presents our empirical study of performance bugs in Android applications and our static analysis technique for automatically detecting common patterns of performance bugs. These two chapters are self-contained, including their own research questions, evaluation, and related work discussions and comparisons. Chapter 5 concludes our completed research work and discusses our ongoing and future work on diagnosing energy efficiency and performance for mobile applications.

Chapter2

Preliminaries

Android is an open-source Linux-based operating system [2]. It is now one of the most widely adopted mobile computing platforms. Many mobile device manufacturers (e.g., Samsung) customize their own Android variants by modifying the Android software stack (e.g., kernel and libraries). Applications running on the Android platform are mostly written in Java language. For performance considerations, developers may write critical parts of their applications using native-code languages such as C and C++. Typically, an Android application is first compiled to Java virtual machine compatible .class files that contain Java bytecode instructions. These .class files are then converted to Dalvik virtual machine executable .dex files that contain Dalvik bytecode instructions. Finally, the .dex files are encapsulated into an Android application package file (i.e., an .apk file) for distribution and installation [1].

2.1 Application Component and Lifecycle

An Android application typically comprises four types of components [1]:

- **Activities.** Activities are the only components that allow graphical user interfaces (GUIs). An application may use multiple activities to provide cohesive user experiences. The GUI layout of each activity component is specified in the activity's layout configuration file.

- **Services.** Services are components that run at background for conducting long-running tasks like sensor data reading. Activities can start and interact with services.

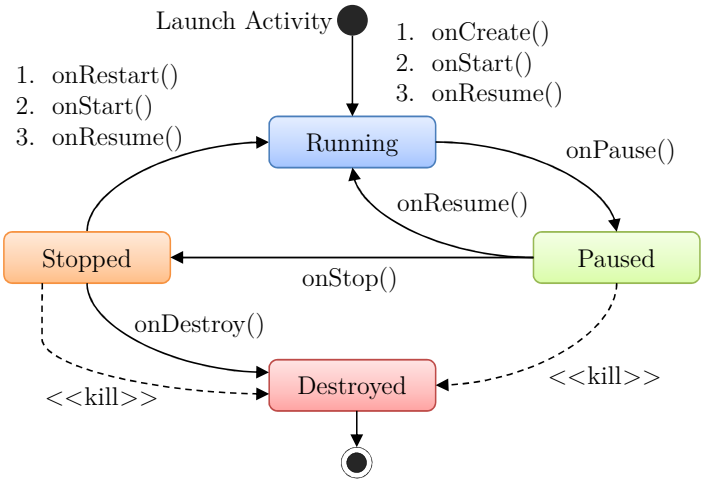

Figure 2.1: The lifecycle of an activity component

- **Broadcast receivers.** Broadcast receivers define how an application responds to system-wide broadcasted messages. It can be statically registered in an application's configuration file (i.e., the AndroidManifest.xml file associated with each application), or dynamically registered at runtime by calling certain Android platform APIs.

- **Content providers.** Content providers manage shared application data, and provide an interface for other components or applications to query or modify these data.

Each application component is required to follow a prescribed lifecycle that defines how this component is created, used, and destroyed. Figure 2.1 shows an activity's lifecycle [1]. It starts with a call to `onCreate()` handler, and ends with a call to `onDestroy()` handler. An activity's foreground lifetime starts after a call to `onResume()` handler, and lasts until `onPause()` handler is called, when another activity comes to foreground. An activity can interact with its users only when it is at foreground. When it goes to background and becomes invisible, its `onStop()` handler would be called. When the users navigate back to a paused or stopped activity, that activity's `onResume()` or `onRestart()` handler would be called, and the activity would come to foreground again. In exceptional cases, a paused or stopped activity may be killed for releasing memory to other applications with higher priorities.

```
LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
LocationListener listener = new LocationListener(){
  public void onLocationChanged(Location loc){
    //do something to use the obtained location data
  }
  public void onStatusChanged(String provider, int status, Bundle extras){}
  public void onProviderEnabled(String provider){}
  public void onProviderDisabled(String provider){}
};
//register the location listener to use GPS sensor
lm.requestLocationUpdate(LocationManager.GPS_PROVIDER, 0, 0, listener);
//unregister location listener when the GPS is no longer needed
lm.removeUpdates(listener);
```

Figure 2.2: GPS sensor usage example

2.2 Single Thread Policy

When an Android application starts, Android OS creates a "main thread" (also known as an "UI thread") to instantiate this application's components. This thread dispatches system calls to responsible application components, and user events to appropriate UI widgets (e.g., buttons). After dispatching, the corresponding components' lifecycle handlers and UI widgets' GUI event handlers will run in the main thread to handle the system calls or user events. This is known as the "single thread policy" [1]. The policy requires developers to control workloads of their applications' main threads (e.g., not overwhelming a main thread with intensive work). Otherwise, applications can easily exhibit poor responsiveness. If an application keeps being unresponsive to users' interactions for a period of time (e.g., 5 seconds), the Android OS will display an "Application Not Responding" (ANR) dialog and offer users an option to force close the application.

2.3 Using Sensors on Android Platforms

Most Android devices have built-in sensors that can help measure users' various environmental conditions (e.g., current location). To use a sensor, an application needs to register a listener with the Android OS and specify a sensing rate. The listener defines how an application reacts to sensor value or status change. When a

```
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
WakeLock wl = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK);
wl.acquire();
if(wl != null){
  //start long running critical computation here
}
//release wake lock when critical computation ends
if(wl.isHeld()){
  wl.release();
}
```

Figure 2.3: Wake lock usage example

sensor is no longer needed, its listener should be unregistered to avoid wasted sensing operations that may drain device battery power [1]. Figure 2.2 give an example usage of GPS sensor. The `onLocationChange()` handler in the location listener class defines how the application uses the location data obtained by the device's GPS sensor. The other three handlers defines how the application reacts to the status change of the GPS sensor (e.g., `onProviderDisabled()` defines how the application reacts when GPS sensor is disabled). The `requestedLocationUpdate()` API call registers the location listener with the Android OS. The second and third parameters of the `requestLocationUpdate()` API specify the minimum time interval and minimum change in distance between location updates, respectively (i.e., configuring the sensing rate). Here in our example, setting them both to zero requests location updates as frequently as possible. When the location sensing is no longer needed, the application should call `removeUpdate()` API to unregister the location listener. Other sensors (e.g., accelerometer) can be used in similar ways, but through the `SensorManager` class. It is worth noting that a slight difference between using GPS sensor and other sensors is that to use a device's GPS sensor, an application needs to declare the permission `android.permission.ACCESS_FINE_LOCATION`.

2.4 Wake Lock Mechanism

Mobile devices have limited battery power, but are usually equipped with energy-consumptive hardwares such as HD screens and WiFi NICs. Extensive use of such

Table 2.1: Different wake levels of wake locks

Wake level	CPU	Screen	Keyboard
PARTIAL_WAKE_LOCK	On	Off	Off
SCREEN_DIM_WAKE_LOCK*	On	Dim	Off
SCREEN_BRIGHT_WAKE_LOCK*	On	Bright	Off
FULL_WAKE_LOCK*	On	Bright	Bright

hardwares can quickly deplete a fully-charged phone battery. To prolong battery life, many mobile computing platforms including Android choose to put energy-consumptive hardwares into a sleep mode (e.g., turning screen off) after a short period of user inactivity. However, there are certain scenarios where an application needs to keep device awake to perform long running critical computation. To cope with this demand, different platforms employ different mechanisms to allow developers to explicitly control the power state of a mobile device. On Android platforms, developers can make use of the wake lock mechanism.

To use a wake lock, an Android application needs to declare the `android.permission.WAKE_LOCK` permission in its configuration file (`AndroidManifest.xml`). After obtaining the permission, it can simply create a `PowerManager.WakeLock` object and specify an intended wake level (see Figure 2.3 for an example). Table 2.1 lists the four wake levels defined by the Android framework.[1] Each wake level has a different effect on the device's power state. For instance, a `PARTIAL_WAKE_LOCK` will ensure that the device CPU is running, but the display and keyboard backlight (if there is a physical keyboard) will be allowed to go off. After creating the wake lock object, an application can then use the APIs on the wake lock to acquire or release the wake lock. Once acquired, the wake locks will have long lasting effects until they are released. Therefore, developers should carefully manage wake locks to avoid energy waste.

[1]Android documentation suggests developers to set the KEEP_SCREEN_ON flag when defining activity components rather than using the wake locks marked by "*" in Table 2.1. The OS will then keep the screen on when the activities are visible to users. Although this avoids manual wake lock management, to precisely control when screens can go off, developers still often use wake locks.

Chapter3

Understanding and Diagnosing Energy Bugs

In this chapter, we first present a large-scale empirical study of real energy bugs from popular Android applications in Section 3.1. The empirical study identified two common patterns of energy bugs. With the findings, we then present a state-based dynamic analysis technique for automatically detecting these patterns of energy bugs in Section 3.2. Next, we introduce our tool implementation and evaluate our technique using real application subjects in Section 3.3. After that, we discuss existing studies related to energy bug diagnosis in mobile applications in Section 3.4. Finally, we summarize this chapter in Section 3.5. The content of this chapter is based on two research papers [65, 68].

3.1 Understanding Energy Bugs

In this section, we conduct an empirical study of real-world energy bugs in Android applications. For ease of presentation, we may use "energy problems" and "energy bugs" interchangeably. Our study aims to answer the following three research questions:

- **RQ1 (Problem magnitude):** *Are energy problems in Android applications serious? Do the problems have a severe impact on smartphone users?*
- **RQ2 (Diagnosis and bug-fixing efforts):** *Are energy problems relatively more difficult to diagnose and fix than non-energy problems? What information do developers need in the energy problem diagnosis and fixing process?*
- **RQ3 (Common causes and patterns):** *What are the common causes of energy problems? What patterns can we distill from them to enable automated diagnosis of these problems?*

3.1.1 Study Methodology

To study these research questions, we first selected a set of commercial Android applications that suffered from energy problems. We randomly collected 608 candidates from Google Play store [19] using a web crawling tool [9]. These applications have release logs containing at least one of the following keywords: *battery*, *energy*, *efficiency*, *consumption*, *power*, and *drain*. We then performed a manual examination to ensure that these applications indeed had energy problems in the past and developers have fixed these problems in these applications' latest versions (note that we did not have access to the earlier versions containing energy problems). This left us with 229 commercial applications. By studying available information such as category, downloads and user comments, we can answer our research question RQ1. However, these commercial applications alone are not adequate enough for us to study the remaining two research questions. This is because to answer research questions RQ2–3, we need to know all details about how developers fix energy problems (e.g., code revisions, the linkage between these revisions and their corresponding bug reports). As such, we also need to study real energy problems with source code available, i.e., from open-source subjects. To find interesting open-source subjects, we first randomly selected 250 candidates from three primary open-source software hosting platforms: Google Code [18], GitHub [17] and SourceForge [34]. Since we are interested in applications with a certain level of development maturity, we refined our selection by retaining those applications that: (1) have at least 1,000 downloads (popular), (2) have a public bug tracking system (traceable), and (3) have multiple versions (well-maintained). These three constraints left us with 173 open-source subjects. We then manually inspected their code revisions, bug reports, and debugging logs. We found 34 of these 173 subjects have reported or fixed energy problems (details are given in Section 3.1.2).

Table 3.1 lists project statistics for all 402 (173 + 229) subjects studied. We observe that these subjects are all popularly downloaded, and cover different application categories. We then performed an in-depth examination of these subjects to answer our research questions. The whole study involved one undergraduate student

Table 3.1: Statistics of our studied Android applications

Application type	Application availability				Application downloads			Covered categories
	Google Code	GitHub	Source Forge	Google Play	Min.	Max.	Avg.	
34 open-source ones (with energy problems)	27/34	8/34	0/34	29/34	1K[1]~5K	5M[1]~10M	0.49M~1.68M	15/32[2]
139 open-source ones (no reported energy problems)	108/139	26/139	10/139	102/139	1K~5K	50M~100M	0.50M~1.22M	24/32
229 commercial ones (with energy problems)	All apps are available on Google Play Store				1K~5K	50M~100M	0.77M~2.02M	27/32

[1]: 1K = 1,000 & 1M = 1,000,000; [2]: According to Google's classification, there were a total of 32 different categories of Android applications at our study time.

Table 3.2: Top five categories of energy-inefficient commercial Android applications

Category	# of inefficient applications
Personalization	59 (25.8%)
Tools	34 (14.8%)
Brain & Puzzle	15 (6.6%)
Arcade & Action	13 (5.7%)
Travel & Local	11 (4.8%)

and four postgraduate students with a manual effort of about 35 person-weeks. We report our findings in the following subsections.

3.1.2 Problem Magnitude

Our selected 173 open-source Android applications contain hundreds of bug reports and code revisions. From them, we identified a total of 66 bug reports on energy problems, which cover 34 applications. Among these 66 bug reports, 41 have been confirmed by developers. Most (32/41) confirmed bugs are considered to be serious bugs with a severity level ranging from medium to critical. Besides that, we found 30 of these confirmed bugs have been fixed by corresponding code revisions, and developers have verified that these code revisions have indeed solved corresponding energy problems.

On the other hand, regarding the 229 commercial Android applications that suffered from energy problems, we studied their user reviews and obtained three findings. First, we found from the reviews that hundreds of users complained that these applications drained their smartphone batteries too quickly and caused great inconvenience for them. Second, as shown in Table 3.1, these energy problems cover 27 different application categories, which are quite broad as compared to the total number of 32 categories. This shows that energy problems are common to different types of applications. Table 3.2 lists the top five categories for illustration. Third, these 229 commercial applications have received more than 176 million downloads in total. This number is significant, and shows that their energy problems have potentially affected a vast number of users.

Based on these findings, we derive our answer to research question RQ1: *Energy problems are serious. They exist in many types of Android applications and affect many users.*

3.1.3 Diagnosis and Bug-Fixing Efforts

To understand how difficult the diagnosis and fixing of energy problems can be, we studied 25 out of the 30 fixed energy bugs in open-source applications. Five fixed bugs were ignored in our study because we failed to recover the links between their bug reports and corresponding code revisions.[1] Table 3.3 gives the basic information of our studied energy bugs, including: (1) bug ID, (2) severity level, (3) revision in which the bug was fixed, and (4) program size of the inefficient revision. Table 3.4 reports our study findings. For each fixed energy bug, Table 3.4 reports: (1) duration in which the bug report is open, (2) number of revisions for fixing the bug, and (3) number of classes and methods that were modified for fixing the bug. We also studied the 11 ($= 41 - 30$) confirmed but not fixed energy problems in open-source applications since four of the eight concerned applications are still actively maintained. We studied how long their bug reports stayed open as well as the

[1]Our manual examination of the commit logs around the bug fixing dates also failed to find the bug-fixing code revisions.

Table 3.3: The studied energy bugs in open-source Android applications

Application name	Downloads	Issue information			
		Issue no.	Severity	Fixed revision	Inefficient revision size (LOC)
DroidAR[1]	5K[3] ~ 10K	27*	Medium	207	18,106
Recycle Locator	1K ~ 5K	33*	Medium	69	3,241
Sofia Public Transport Nav.	10K ~ 50K	38*	Medium	156	1,443
Sofia Public Transport Nav.	10K ~ 50K	76*	Critical	156	1,649
Google Voice Location[6]	10K ~ 50K	4*	Medium	20	4,632
BitCoin Wallet	10K ~ 50K	86	Medium	1bbc6295083c	27,220
Osmdroid	10K ~ 50K	53*	Medium	751	13,385
Osmdroid	10K ~ 50K	76*	Medium	315	8,636
Zmanim	10K ~ 50K	50/56*	Critical	323	4,807
Transdroid	10K ~ 50K	19*	Medium	Version 0.8.0	11,715
Geohash Droid	10K ~ 50K	24*	Medium	6d8f10153a48	6,682
AndTweet[6]	10K ~ 50K	29*	Medium	4a1f1f9683f2	8,908
K9Mail	1M[3] ~ 5M	574	Medium	933	72,723[5]
K9Mail	1M ~ 5M	864	Medium	317	72,723
K9Mail	1M ~ 5M	1031	Medium	1395	72,723
K9Mail	1M ~ 5M	1643/1694	Medium	1731	72,723
K9Mail	1M ~ 5M	N/A[4]	N/A	4542e64	72,723
Open-GPSTracker[6]	100K ~ 500K	70	Critical	33f6e78aad9a	4,447
Open-GPSTracker[6]	100K ~ 500K	128*	Low	3aa9fb4d4ffb	9,174
Ebookdroid	500K ~ 1M	23*	Medium	138	14,351
CSipSimple	500K ~ 1M	1674	Critical	1386	54,966
c:geo[2]	1M ~ 5M	1709	Critical	cecda72	33,514
BableSink[6]	1K ~ 5K	N/A*	N/A	9fbcbf01ce	1,718
CWAC-Wakeful	1K ~ 5K	N/A*	N/A	c7d440f115	896
Ushahidi[6]	10K ~ 50K	N/A*	N/A	337b48f	10,186

[1]: Applications from DroidAR to CSipSimple are hosted on Google Code.

[2]: Applications from c:geo to CommonsWare are hosted on GitHub. [3]: 1K = 1,000 & 1M = 1,000,000;

[4]: The symbol "N/A" means "unknown", and the corresponding bugs are found by studying commit logs.

[5]: The size of K9Mail is based on revision fdfaf03b7a because we failed to access its original SVN repository after it switched to use Git.

[6]: All applications except Google Voice Location, AndTweet, Open-GPSTracker, BabbleSink and Ushahidi are still actively maintained (continuous code revisions).

Table 3.4: Diagnosis and fixing effort of our studied energy bugs

Application name	Issue no.	Diagnosis and fixing efforts			
		Issue open duration (Days)	# of revisions to fix	# of changed classes	# of changed methods
DroidAR	27*	7	3	4	18
Recycle Locator	33*	1	1	1	5
Sofia Public Transport Nav.	38*	19	2	3	7
Sofia Public Transport Nav.	76*	1	1	1	1
Google Voice Location	4*	330	10	4	37
BitCoin Wallet	86	30	1	2	4
Osmdroid	53*	243	1	1	4
Osmdroid	76*	11	1	1	5
Zmanim	50/56*	35	1	6	14
Transdroid	19*	9	1	1	7
Geohash Droid	24*	3	1	1	6
AndTweet	29*	240	1	6	22
K9Mail	574	101	1	2	9
K9Mail	864	49	3	6	8
K9Mail	1031	20	1	1	1
K9Mail	1643/1694	6	2	3	2
K9Mail	N/A	N/A	1	1	2
Open-GPSTracker	70	2	1	3	9
Open-GPSTracker	128*	9	5	7	8
Ebookdroid	23*	2	1	4	5
CSipSimple	1674	6	1	1	1
c:geo	1709	16	1	2	9
BableSink	N/A*	N/A	1	1	1
CWAC-Wakeful	N/A*	N/A	1	1	1
Ushahidi	N/A*	N/A	1	2	9

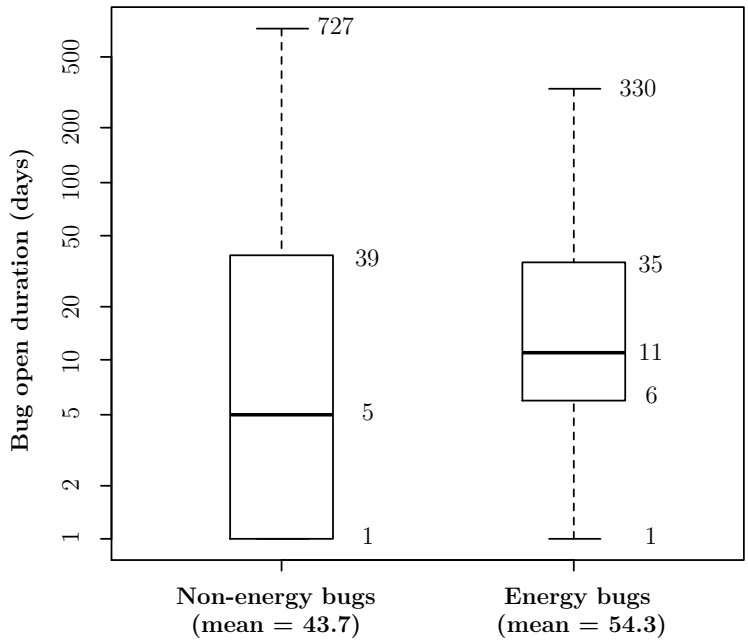

Figure 3.1: Bug open duration of energy and non-energy bugs

duration of their related discussions. From these studies, we made the following three observations.

First, 24 out of the 25 energy problems listed in Tables 3.3 and 3.4 are serious problems whose severity ranges from medium to critical. Developers take, on average, 54 work-days to diagnose and fix them. For comparison, we checked the remaining 1,967 non-energy bugs of similar severity (i.e., medium to critical) reported on these applications before March 2013. We found that these non-energy bugs were fixed, on average, within 43 work-days. Figure 3.1 gives a detailed box plot of open duration for the energy and non-energy bugs we studied. For example, the median open duration for non-energy bugs is five days while the median open duration for energy bugs is 11 days. Such comparison results suggest that energy problems are likely to take a longer time to fix. We further conducted a Mann-Whitney U-test [71] of the following two hypotheses:

- **Null hypothesis H_0.** Fixing energy problems does not take a significantly longer time than fixing non-energy problems.

- **Alternative hypothesis H_1.** Fixing energy problems takes a significantly longer time than fixing non-energy problems.

Our test results show that the p-value is 0.0327 (< 0.05), indicating that the null hypothesis H_0 can be rejected with a confidence level of over 0.95. Therefore, we can conclude that energy problems take a relatively longer time to fix.

Second, for the 11 confirmed but not fixed energy problems, we found that developers closed five of them because they failed to reproduce corresponding problems and they did not receive user complaints after some seemingly irrelevant code revisions. For three of the remaining six problems, we found that developers are still working on fixing them without success [10, 23, 25]. Their three associated bug reports have been remained open for more than two years. For example, CSipSimple is a popular application for video calls over the Internet. Developers have discussed its energy problem (issue 81) tens of times, trying to find the root cause, but failed to make any satisfactory progress so far. Due to this, some disappointed users uninstalled CSipSimple, as indicated from their comments on the bug report [10].

Third, as shown in Table 3.4, in 21 out of 25 cases, developers fixed the reported energy problems in one or two revisions. These fixes require non-trivial effort. For example, 16 out of these 25 fixes require modifying more than 5 methods. On average, developers fixed these 25 problems by modifying 2.6 classes and 7.8 methods.

We also looked into discussions on fixed energy bugs. We found that many of these bugs are intermittent. Developers generally consider these intermittent bugs as complex issues. In order to reproduce them, developers have to know details about how users interact with their applications before these problems occur. Developers often have to analyze debugging information logged at runtime in order to identify the root causes of these problems. For example, to facilitate energy waste diagnosis, K9Mail developers gave special instructions on how users could provide useful debugging logs [23]. This may become additional overhead for smartphone users when they report energy problems.

Based on these findings, we derive our answer to research question RQ2: *It is relatively more difficult to diagnose and fix energy problems, as compared to non-energy problems; user interaction contexts and debugging logs can help problem diagnosis, but they require additional user-reporting efforts, which may not be desirable.*

3.1.4 Common Patterns of Energy Bugs

Energy inefficiency is a non-functional issue whose causes can be complex and application-specific. For example, CSipSimple issue 1674 [10] happened because the application monitored too many broadcasted messages, and its issue 744 was caused by unnecessary talking with a verbose server [10]. Nevertheless, by studying the bug-fixing patches and bug report comments of the earlier mentioned 25 fixed energy problems, we observe that 16 of them (64.0%) are due to misuse of sensors or wake locks. These problems are marked with "*" in Tables 3.3 and 3.4.

To confirm that misuse of sensors or wake locks can indeed lead to energy problems in Android applications, we analyzed the API usage of all 402 applications. On the Android platform, applications need to call certain APIs to invoke system functionalities. For example, an application needs to call the `PowerManager.WakeLock.acquire()` API to acquire a wake lock from Android OS so as to keep a device awake for computation (see Section 2.4 for more details). As such, API usage analysis can disclose which Android features are being used by an application. To analyze API usage of our 173 open-source applications, we compiled their source code to obtain Java bytecode. For commercial applications, we handled them differently. We first downloaded their .apk files from Google Play store using an open-source tool Real APKLeecher [29].[2] We then transformed their Dalvik bytecode (contained in the .apk files) to Java bytecode using dex2jar [11], a popular Dalvik bytecode retargeting tool [78]. Finally, we scanned the Java bytecode of each application to analyze their API usage. From the analysis, we obtained two major findings. First, 46.7% (14/30) open-source applications that use sensors and 68.0% (17/25) open-source applications that acquire wake locks were confirmed to have energy problems. Second, 51.1% (117/229) energy inefficient commercial applications use sensors or wake lock. These findings suggest that misuse of sensors or wake locks could be closely associated with energy problems in Android applications.

[2]The original Real APKLeecher is GUI-based. We modified it to support command line usage for study automation. The modified version can be obtained at: `http://sccpu2.cse.ust.hk/greendroid`.

AndTweet Issue 29: *"Issue 29 is due to the design of AndTweetService: It starts right after boot and acquires a partial wake lock. According to the Android documentation, the acquired wake lock ensures that the CPU is always running. The screen might not be on. This is why few users had noticed the issue before."*

Geohash Droid Issue 24: *"GeohashService should slow down its GPS updates to one every thirty seconds if nothing besides the notification bar is waiting for updates."*

Figure 3.2: Developers' comments on energy bugs

Based on these findings, we further studied the discussions on fixed energy problems and their bug-fixing patches. We then observed two types of coding phenomena concerning sensor or wake lock misuse that can lead to serious energy waste in Android applications:

Pattern 1: Missing sensor or wake lock deactivation. To use a sensor, an application needs to register a listener with Android OS, and specify a sensing rate [1]. The listener defines how an application reacts to sensor value or status changes. When a sensor is no longer needed, its listener should be unregistered in time. As stated in Android documentation, forgetting to unregister sensor listeners can lead to unnecessary sensing operations that waste battery energy. Similarly, to keep a smartphone awake for computation, an application needs to acquire a wake lock from Android OS and specify a wake level. For example, a full wake lock can keep a phone's CPU awake and its screen on at full brightness. The acquired wake lock should be released as soon as the computation completes. Forgetting to release wake locks in time can quickly drain a phone's battery [1]. For example, Figure 3.2 gives a developer's comment on an energy problem in AndTweet, a Twitter client [4]. AndTweet starts a background service `AndTweetService` right upon receiving a broadcast message indicating that Android OS has finished booting. When `AndTweetService` starts, it acquires a partial wake lock, which is not released until `AndTweetService` is destroyed. However, due to a design defect, `AndTweetService` keeps running at background, unless it encounters an external storage exception (e.g., SD card being un-mounted) or is killed explicitly by users, while such cases

are rare. As a result, AndTweet can waste a surprisingly large amount of battery energy due to this missing wake lock deactivation problem.[3]

Pattern 2: Sensory data underutilization. Sensory data are acquired at the cost of battery energy. These data should be effectively used by applications to produce perceptible benefits to smartphone users. However, when an application's program logic becomes complex, sensory data may be "underutilized" in certain executions. In such executions, the energy cost for acquiring sensory data may outweigh the actual usages of these data. We call this phenomenon "sensory data underutilization". We observed that sensory data underutilization often suggests design or implementation defects that can cause energy waste. For example, Figure 3.3(a) gives the concerned code snippet of a location data underutilization problem in an entertainment application Geohash Droid. This application is designed for users who like adventures. It randomly selects a location for users and navigates them there using GPS sensors. As the code in Figure 3.3(a) shows, Geohash Droid maintains a long running `GeohashService` at background for location sensing. `GeohashService` registers a location listener with Android OS when it starts (Lines 7–16), and unregisters the listener when it finishes (Lines 22–25). Once it receives location updates, it refreshes the smartphone's notification bar (Line 11), which provides users with quick access to their current locations. After that, it notifies remote listeners (e.g., the navigation map) to use updated location data (Lines 12, 27–36). Thus, location data are used to produce perceptible benefits to users when remote listeners are actively listening to such location updates. However, there are chances when no remote listeners are alive (e.g., the navigation map will not be alive when it loses user focus). When this happens, Geohash Droid would keep receiving the phone's GPS coordinates, simply for updating its notification bar [16]. Such updates do not reflect effective use of newly captured GPS coordinates, while the battery's energy is continuously consumed. Geohash Droid developers received a lot of user complaints for such battery drain. After intensive discussions, developers identified the cause

[3]For more details, readers can refer to the following classes in package `com.xorcode.andtweet` of application AndTweet-0.2.4: `AndTweetService`, `AndTweetServiceManager`, `TimelineActivity` and `TweetListActivity` [4].

```
1. public class GeohashService extends Service {
2.   private ArrayList<RemoteListener> mListeners;
3.   private LocationManager lm;
4.   private LocationListener gpsListener;
5.   public void onStart(Intent intent, int StartId){
6.     mListeners = new ArrayList<RemoteListener>();
7.     //get a reference to system location manager
8.     lm = getSystemService(LOCATION_SERVICE);
9.     gpsListener = new LocationListener() {
10.      public void onLocationChanged(Location loc) {
11.        updateNotificationBar(loc);
12.        notifyRemoteListeners(loc);
13.      }
14.    };
15.    //GPS listener registration
16.    lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
17.  }
21.  //more code from GeohashService
22.  public void onDestroy() {
23.    //GPS listener unregistration
24.    lm.removeUpdates(gpsListener);
25.  }
26.  //notify alive remote listeners for loc change
27.  public void notifyRemoteListeners(Location loc){
28.    final int N = mListeners.size();
29.    for(int i = 0; i < N; i++) {
30.      RemoteListener listener = mListeners.get(i);
31.      if(listener.isAlive()){
32.        //remote listeners consume location data
33.        listener.locationUpdate(loc);
34.      }
35.    }
36.  }
37. }
```

(a) Example from the Geohash Droid application (Issue 24)

```
1. public class MapActivity extends Activity {
2.   private Intent gpsIntent;
3.   private BroadcastReceiver myReceiver;
4.   public void onCreate(){
5.     gpsIntent = new Intent(GPSService.class);
6.     startService(gpsIntent); //start GPSService
7.     myReceiver = new BroadcastReceiver() {
8.       public void onReceive(Intent intent) {
9.         LocData loc = intent.getExtra();
10.        updateMap(loc);
11.        if(trackingModeOn) persistToDatabase(loc);
12.      }
13.    }
14.    //register receiver for handling loc change messages
15.    IntentFilter filter = new IntentFilter("loc_change");
16.    registerReceiver(myReceiver, filter);
17.  }
18.  public void onDestroy() {
19.    //stop GPSService and unregister broadcast receiver
20.    stopService(gpsIntent);
21.    unregisterReceiver(myReceiver);
22.  }
23. }
31. public class GPSService extends Service {
32.   private LocationManager lm;
33.   private LocationListener gpsListener;
34.   public void onCreate(){
35.     //get a reference to system location manager
36.     lm = getSystemService(LOCATION_SERVICE);
37.     gpsListener = new LocationListener() {
38.       public void onLocationChanged(Location loc) {
39.         LocData formattedLoc = processLocation(loc);
40.         //create and send a location change message
41.         Intent intent = new Intent("loc_change");
42.         intent.putExtra("data", formattedLoc);
43.         sendBroadcast(intent);
44.       }
45.     };
46.     //GPS listener registration
47.     lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
48.   }
49.   public void onDestroy() {
50.     //GPS listener unregistration
51.     lm.removeUpdates(gpsListener);
52.   }
53. }
```

(b) Example from the Osmdroid application (Issue 53)

Figure 3.3: Motivating examples for sensory data underutilization energy bugs

of this problem and chose to reduce the GPS sensing rate when there is no active remote listener for such location updates. Figure 3.2 shows their comment after fixing this energy problem.

Another interesting example is the energy problem in Osmdroid, a popular map-based navigation application. Figure 3.3(b) gives a simplified version of the concerned code. The application has three components: (1) `MapActivity` for displaying a map to its users, (2) `GPSService` for location sensing and data processing in background, and (3) a broadcast receiver for handling location change messages (Lines 7–13). When `MapActivity` is launched, it starts `GPSService` (Lines 5–6), and registers its broadcast receiver (Lines 15–16). `GPSService` then registers a location listener with the Android OS when it starts (Lines 36–47). When the application's users change their locations (e.g., during a walk), `GPSService` would receive and process new location data (Line 39), and broadcast a message with the processed data (Lines 41–43). The broadcast receiver would then use the new location data to refresh a map (Line 10). If the users have enabled location tracking, these location data would also be stored to a database (Line 11). If the Android OS plans to destroy `MapActivity` (Lines 18–22), `GPSService` would be stopped (Line 20), and both the location listener and broadcast receiver would be unregistered (Lines 21, 51). These all work seemingly smoothly. However, if Osmdroid's users switch from `MapActivity` to any other activity, `MapActivity` would be put to background (not destroyed), but `GPSService` would still keep running for location sensing. If the location tracking functionality is not enabled, all collected location data would be used to refresh an invisible map. Then, a huge amount of energy would be wasted [28]. To fix this problem, developers chose to disable the GPS sensing conditionally (e.g., according to whether the location tracking mode is enabled or not), when `MapActivity` goes to background.

From the preceding two examples of sensory data underutilization, we make three observations. First, locating sensory data underutilization problems can provide desirable opportunities for optimizing an application's energy consumption. When such problems occur, the concerned application can deactivate related sensors or tune down their sensing rates to avoid unnecessary energy cost. Second, to detect

such sensory data underutilization problems, one should track how sensory data are transformed into different forms of program data and consumed in different ways. Third, sensory data underutilization problems may occur only at certain application states. For example, Geohash Droid wastes energy only when there is no active remote listener waiting for location updates. In Osmdroid, if its user has enabled the location tracking functionality before `MapActivity` goes to background, even if it is consuming non-trivial energy due to continuous GPS sensing, we cannot simply consider this as energy waste. This is because the collected location data could be stored for future uses, producing perceptible user benefits afterwards. These three observations motivate us to consider a state-based approach to analyzing sensory data utilization for Android applications. Such analysis can help developers judge whether their applications are using sensory data in a cost-effective way and provide optimization opportunities for energy efficiency if necessary.

3.1.5 Threats to Validity

The validity of our empirical study may be subject to some threats. One is the representativeness of our selected Android applications. To minimize this threat and avoid subject selection bias, we selected 173 open-source and 229 commercial Android applications spanning 27 different categories. These applications have been popularly downloaded and can be good representatives of real-world Android applications. Another potential threat is the manual inspection of our selected subjects. We understand that this manual process may be error-prone. To reduce this threat, we have all our data and findings independently inspected by at least two researchers. We cross-validated their inspection results for consistency.

3.2 Energy Efficiency Analysis

In this section, we elaborate on our energy efficiency analysis approach. We will begin with an overview of our approach.

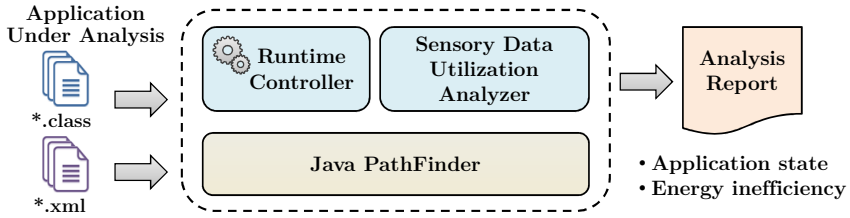

Figure 3.4: Overview of our energy efficiency analysis approach

3.2.1 Approach Overview

Our energy efficiency diagnosis is based on dynamic information flow analysis [59]. Figure 3.4 shows its high-level abstraction. It takes as inputs the Java bytecode and configuration files of an Android application. The Java bytecode defines the application's program logic, and can be obtained by compiling the application's source code or transforming its Dalvik bytecode [78]. The configuration files specify the application's components, GUI layouts, and so on. The general idea of our diagnosis approach is to execute an Android application using Java PathFinder (JPF) [93], a widely-used dynamic model checker for general Java programs (recall that Android applications are essentially Java programs), to systematically explore its application states. During the execution, our approach monitors all sensor registration/unregistration and wake lock acquisition/releasing operations. It feeds mock sensory data to the application when related sensor listeners are properly registered. It then tracks the propagation of these sensory data as the application executes, and analyzes how they are utilized at different application states. At the end of the execution, our approach compares sensory data utilization across explored states, and reports those states where sensory data are underutilized. It also checks which sensor listeners are forgotten to be unregistered, and which wake locks are forgotten to be released, and reports these anomalies.

The above high-level abstraction looks straightforward, but contains some challenging questions: *How can one execute an Android application and systematically explore its states, especially in JPF? How can one identify those executions that involve sensory data? How can one measure and compare sensory data utilization*

at application states explored by these executions? We answer these questions in the following subsections.

3.2.2 Application Execution and State Exploration

Android applications are mostly designed to interact with smartphone users. Their executions are often triggered by user interaction events. Typically, an Android application starts with its main activity, and ends after all its components are destroyed. During its execution, the application keeps handling received user interaction events and system events (e.g., broadcasted events) by "calling" their handlers according to Android specifications.[4] Each call to an event handler may change the application's state by modifying its components' local or global program data. As such, in order to execute an application and explore its state space in JPF, we need to: (1) generate user interaction events, and (2) guide JPF to schedule corresponding event handlers.

Before going into the technical details, we first formally define our problem domain and clarify our concept of *bounded state space exploration*. We use P to denote the Android application under diagnosis, and E to denote the set of possible user interaction events for this application.

Definition 1 (User interaction event sequence): A *user interaction event sequence* $\overline{seq} = [e_1, e_2, \ldots, e_n]$, where each $e_i \in E$ is a user interaction event. Operation $len(\overline{seq})$ returns the length of the sequence $\overline{seq}$, and operation $head(\overline{seq}, k)$ returns a subsequence with the first k user interaction events in $\overline{seq}$. We denote the set of all possible user interaction event sequences as SEQ.

The SEQ set is theoretically unbounded as users can interact with an application in infinite ways.

Definition 2 (Application execution): An *execution* t of application P is triggered by a sequence of user interaction events $\overline{seq}$. We denote such an execution

[4]Android applications are event-driven. Their program code comprises many loosely coupled event handlers, among which no explicit control flow is specified. At runtime, these event handlers are in fact called by the Android framework, which builds upon hundreds of native library classes.

as $t = exec(P, \overline{seq})$. Then the set of all possible executions T for the application P is:

$$T = \{exec(P, \overline{seq}) \mid \overline{seq} \in SEQ\}.$$

Definition 3 (State and state space):[5] During its execution, application P's state changes from s_0, which is P's initial state, to s' after it handles a sequence of user interaction events $\overline{seq}$, where $len(\overline{seq}) \geq 1$. We represent the new state s' as $\langle s_0, \overline{seq} \rangle$. Then we can define the state space explored for application P during its execution $t = exec(P, \overline{seq})$ as:

$$S_t = \{\langle s_0, head(\overline{seq}, k) \rangle \mid 1 \leq k \leq len(\overline{seq})\}.$$

As SEQ is unbounded, there exist an infinite number of different executions for an application, that is, set T is also unbounded. Therefore, we have to restrict total execution times and state space exploration in our diagnosis. We then define our bounded state space exploration, in which we control the length of user interaction event sequences.

Definition 4 (Bounded state space exploration): Given a bound value b (≥ 1) on the length of user interaction event sequences, our diagnosis examines the following executions for an Android application P:

$$T_b = \{exec(P, \overline{seq}) \mid \overline{seq} \in SEQ \ \& \ len(\overline{seq}) \leq b\}.$$

For these executions, our diagnosis explores the following space of states:

$$S_b = \bigcup_{t \in T_b} S_t$$

After defining the bounded state space exploration concept, we proceed to introduce our diagnosis approach. To effectively explore an Android application's

[5]We discuss state changes at an event handling level as users have control on that. We do not consider finer-grained state changes or state equivalence in our work.

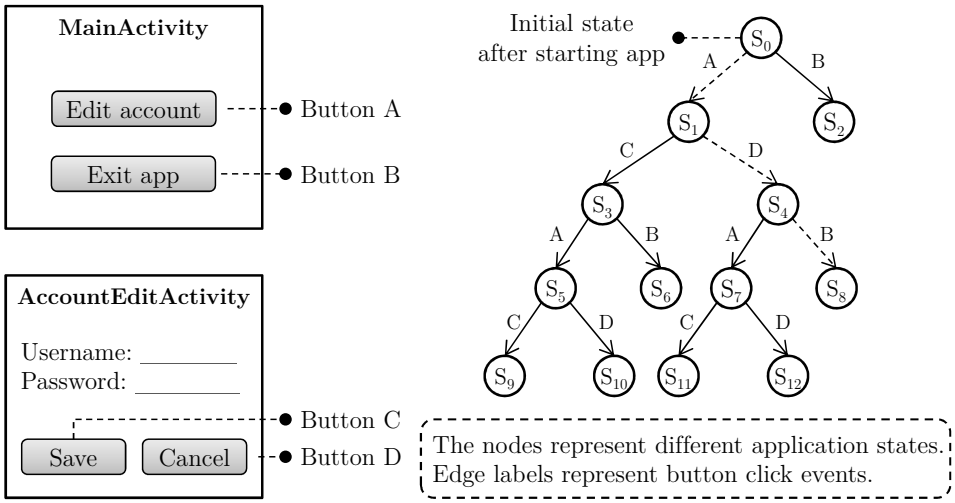

Figure 3.5: Illustration of event sequence generation

state space, we need to generate event sequences of user interactions and schedule corresponding event handlers. We address these two technical issues below.

Event sequence generation. Our runtime controller, as illustrated in Figure 3.4, simulates user interactions by generating corresponding event sequences. Conceptually, the generation process contains two parts: static and dynamic. In the static part, i.e., before executing an application, we first analyze the application's configuration files to learn the GUI layouts of its activity components (recall that only activities have GUIs). Specifically, we map each GUI widget (e.g., a button) of an activity component to a set of possible user actions (e.g., button clicks). This constructs a *user event set* for each activity. In the dynamic part, i.e., when executing an application, our runtime controller monitors the application's execution history and current state. When the application waits for user interactions (e.g., after an activity's `onResume()` handler is called), our controller would generate required events and feed them to the foreground activity for handling. This is done in an exhaustive way by enumerating all possible events associated with each activity component. Our controller continues doing so until the length of a generated event sequence reaches the required upper bound or the application exits. In this way, we generate all possible event sequences bounded by a length limit b, and explore its corresponding bounded state space S_b. For ease of understanding, we provide an example to illustrate the event sequence generation process.

The example application in Figure 3.5 contains two activities: `MainActivity` and `AccountEditActivity`. When this application starts, `MainActivity` would appear first. Its users can click the "Edit account" button to edit their account information in another `AccountEditActivity`'s window (`MainActivity` would then be put to background). After editing, users can save the changes by clicking the "Save" button or discard the changes by clicking the "Cancel" button. This also brings users back to the previous `MainActivity`'s window (`AccountEditActivity` would then be destroyed). To exit the application, the users can click the "Exit app" button in the `MainActivity`'s window. For ease of presentation, suppose that: (1) we consider only button click events (our tool implementation can handle other types of events, e.g., filling textboxes and selecting from dropdown lists), (2) the event sequence length bound is set to four, and (3) each generated event is correctly handled (e.g., after clicking "Exit app", the application indeed exits).

Based on these assumptions, we consider generating event sequences for this example application. Our controller first constructs user event sets for the two activities. For instance, the user event set for `MainActivity` is {click "Edit account" button, click "Exit app" button}. At runtime, when `MainActivity` waits for user interactions, our controller can enumerate and generate all events in `MainActivity`'s user event set in turn. If it generates an "Edit account" button click event, `AccountEditActivity` would come to foreground. When `AccountEditActivity` is ready for user interactions, our controller similarly enumerates and generates all events in `AccountEditActivity`'s user event set in turn. This event generation process continues until the length of a generated event sequence reaches four or the application exits (e.g., when the "Exit app" button is clicked). The tree on the right of Figure 3.5 illustrates this event sequence generation process. The nodes on the tree represent different application states and the labels on edges that connect the nodes represent button click events. Each path from the root node to a leaf node corresponds to one user interaction event sequence. For example, the path with dashed edges represents an event sequence of length three (the first application starting event is not counted): starting the application, clicking "Edit account" button, clicking

"Cancel" button, and finally clicking "Exit app" button.[6] Other sequences can be explained similarly.

Event handler scheduling. With event sequences generated to represent user interactions, we now consider how to schedule event handlers properly. As mentioned earlier, Android applications consist of a set of loosely-coupled event handlers, among which no explicit control flow is specified. Existing analysis techniques for Android applications commonly assume that developers should specify calling relationships between these event handlers [84]. However, this is not practical. Real-world Android applications typically contain hundreds of event handlers (e.g., the application DroidAR used in our evaluation has 149 event handlers). Manually specifying calling relationships between these event handlers is labor-intensive and error-prone. Therefore, in this work we do not make such an assumption. Instead, we propose to derive an application execution model (or AEM) from Android specifications, and leverage it to guide the runtime scheduling of event handlers. The extracted AEM model plays the role of enforcing calling relationships between event handlers. Specifically, the AEM model is a collection of temporal rules that are prescribed by the Android framework and followed by all Android applications (i.e., such rules are application-generic). We define the model as follows:

$$AEM = \{R_i \mid R_i \; is \; a \; temporal \; rule \; of \; form \; [\psi], [\phi] \Rightarrow \lambda\}$$

In each rule R_i, symbols ψ and λ represent two temporal formulae expressed in linear-time temporal logic. They make assertions about the past and future, respectively. Symbol ϕ represents a propositional logic formula making assertions about the present. Specifically, ψ describes what has happened in history during an application execution, ϕ evaluates the current situation (e.g., what system or user event is received), and λ claims what is expected. Therefore, the whole rule expresses the meaning: *If both ψ and ϕ hold, λ is expected.*

[6] In our implementation, the "start application" and "exit application" events are by defaults generated. That means each generated event sequence starts with the "start application" events and ends with the "exit application" events. Besides, there is no standard way to exit an Android application, our way is to destroy all active activity and service components.

Table 3.5: Example temporal rules in our AEM model

Rule 1: When should an activity's lifecycle handler act.onStart() be called? $[X^{-1}\ act.onCreate()], [\neg\ ACT\_FINISH\_EVENT] \Longrightarrow X\ act.onStart()$
Rule 2: When should GUI widget's click event handler view.onClick() be called? $[(\neg act.onPause()\ S\ act.onResume() \wedge (\neg \text{view}.reg(null)\ S\ \text{view}.reg(listener))],$ $[VIEW\_CLICK\_EVENT] \Longrightarrow X\ listener.onClick()$
Rule 3: When should a dynamic message handler rcv.onReceive() be called? $[\neg \text{rcv}.unreg()\ S\ \text{rcv}.reg()], [MSG\_EVENT] \Longrightarrow X\ \text{rcv}.onReceive()$
Rule 4: When should a static message handler Receiver.onReceive() be called? $[True], [MSG\_EVENT] \Longrightarrow X\ Receiver.onReceive()$

We give some examples of temporal rules in Table 3.5. For the entire collection of 29 rules, readers may refer to our research paper [69].[7] In these example rules, propositional connectives like $\wedge$, $\Rightarrow$, and $\neg$ follow their traditional interpretations, i.e., conjunction, implication, and negation. For temporal connectives, we follow Etessami et al.'s notation [49], which is explained in the following. Unary temporal connective X means "next", and its past time analogue X^{-1} means "previously". Binary temporal connective S means "since". Specifically, a temporal formula "$F_1\ S\ F_2$" means that F_2 held at some time in the past, and since then F_1 always holds.

We give explanation for the rules in Table 3.5. The first rule states that an activity's `onStart()` handler is to be called after its `onCreate()` handler completes as long as this activity is not forced to finish. The second rule states that a GUI widget's click event handler is to be called if: (1) the widget (e.g., a button) is clicked, (2) its enclosing activity is at foreground (i.e., the activity's `onPause()` handler has not been called since the last call to its `onResume()` handler), and (3) its click event listener is properly registered. The third rule disables the call to a message event handler before its registration and after its unregistration. The last

[7] We do not claim the completeness of the AEM model. We will show in our later evaluation that the current version of our AEM model already suffices for verifying many real-world Android applications.

rule states that a static message event handler is to be called upon any broadcasted message.

Our AEM model, i.e., the collection of 29 temporal rules, is converted to a decision procedure which determines the event handlers to be called in the next step according to an application's execution history and its newly received events (events are handled in turn). This event handler scheduling is always deterministic, except when there are multiple receivers registered (either dynamically or statically) for broadcast messages from the same source.[8] If this is the case, the `onReceive()` handlers of those registered receivers are to be called according to the receiver registration orders. By this means, we can exercise an Android application in JPF's Java virtual machine, and systematically explore its state space.

3.2.3 Detecting Missing Sensor or Wake Lock Deactivation

We next discuss how to detect energy problems when exploring an application's state space. As mentioned earlier, missing sensor or wake lock deactivation is one common cause of energy problems. This shares some similarity with traditional resource leak problems, where a program fails to release its acquired system resources (e.g., memory blocks, file handles, etc.) [92]. Resource leak problems can cause system performance degradation (e.g., slower response), and similarly missing deactivation of sensors or wake locks can also waste valuable battery energy. Besides, according to Android process management policy [1], sensors and wake locks are not automatically deactivated even when the application components that activated them are destroyed (e.g., `onDestroy()` handler is called). We will give an example and details in Section 3.3.2. Based on the preceding state exploration efforts, we can now adapt existing resource leak detection techniques [39, 94] to detect missing sensor or wake lock deactivation bugs. In particular, our diagnosis monitors the execution of an Android application and keeps checking the violation of the following two policies:

[8] Although we did not observe such cases in our experiments, registering multiple receivers for broadcast messages from the same source is grammatically acceptable in Android applications.

- **Sensor management policy:** A sensor listener l, once registered, should be unregistered eventually before the application component that registered l is destroyed.
- **Wake lock management policy:** A wake lock wl, once acquired, should be released eventually before the application component that acquired wl is destroyed.

Note that such checking is feasible only after we have addressed the event sequence generation and event handler scheduling problems for Android applications.

3.2.4 Sensory Data Utilization Analysis

During an Android application's execution, its collected sensory data are transformed into different forms and consumed by different application components. We need to track these data usages for energy efficiency analysis. We do it at the bytecode instruction level by dynamic tainting. Our technique contains three phases: (1) tainting each collected sensory datum with a unique mark; (2) propagating taint marks as the application executes; (3) analyzing sensory data utilization at different application states. We elaborate on the three phases in the following.

A. Preparing and tainting sensory data

In the first phase, we generate mock sensory data from an existing sensory data pool, which is controlled with different precision levels. They are then fed to the application under analysis after each event handler call. The object reference to each sensory datum is initialized with a unique taint mark before the datum is fed to the application. The taint mark will be propagated with the datum together for later analysis.

B. Propagating taint marks

At runtime, an Android application's collected sensory data are transformed into different forms by assignment, arithmetic, relational, and logical operations. For example, the Osmdroid application in Figure 3.3(b) has its `loc` object (Line 38)

transformed to another `formattedLoc` object (Line 39), which further affects the `intent` object (Line 42). Later, by message communication, this `intent` object is propagated to a broadcast receiver and converted back to the `loc` object (Line 9), which may or may not affect database content, depending on the variable `trackingModeOn`'s value (Line 11). Such data flows need to be tracked to propagate taint marks so as to identify which program data depend on the collected sensory data. Based on this information, one is then able to analyze sensory data utilization.

Our technique intercepts the execution of a subset of Java bytecode instructions at runtime and propagates taint marks in JPF's Java virtual machine according to our tainting policy.[9] A key advantage of such an instruction-level taint propagation is that it does not require application-specific program instrumentation, which is often time-consuming and error-prone. Table 3.6 gives our tainting policy, which comprises 12 taint propagation rules. These rules handle taint propagations along data dependencies. They are expressed in the following form:

$$T(A) = T(B) \ \cup \ T(C)$$

This means that data B's and C's taint marks are merged to become data A's taint mark. Note that B and C can be optional. Each taint propagation rule in Table 3.6 is designed for a set of bytecode instructions with similar semantics (explained in the lower part of Table 3.6). For example, Rule 6 is for all binary calculation bytecode instructions (totally 37 instructions) such as `fadd` and `iand`. The instruction `fadd` adds two floating numbers popped from the operand stack in the current method call's frame, and pushes the addition result back into this operand stack. Similarly, the instruction `iand` performs a bitwise "and" operation on two integers popped from the operand stack in the current method call's frame, and pushes the operation result back into the stack. For all such binary calculation bytecode instructions, our taint propagation works as follows (Rule 6): the result

[9]On real devices, an Android application runs in a register-based Dalvik virtual machine, while JPF's Java virtual machine is stack-based. This difference does not affect our analysis.

Table 3.6: Taint propagation policy

Index	Instruction type	# insns	Instruction semantics	Taint propagation rule
1	***Const*** *C*	15	$stack[0] \leftarrow C$	$T(stack[0]) = \emptyset$
2	***Load*** *index*	25	$stack[0] \leftarrow localVar_{index}$	$T(stack[0]) = T(localVar_{index})$
3	***LoadArray*** *arrayRef, index*	8	$stack[0] \leftarrow arrayRef\,[index]$	$T(stack[0]) = T(arrayRef) \cup T(arrayRef\,[index])$
4	***Store*** *index*	25	$localVar_{index} \leftarrow stack'[0]$	$T(localVar_{index}) = T(stack'[0])$
5	***StoreArray*** *arrayRef, index*	8	$arrayRef\,[index] \leftarrow stack'[0]$	$T(arrayRef\,[index]) = T(stack'[0])$
6	***Binary-op***	37	$stack[0] \leftarrow stack'[1] \otimes stack'[0]$	$T(stack[0]) = T(stack'[0]) \cup T(stack'[1])$
7	***Unary-op***	20	$stack[0] \leftarrow \ominus\, stack'[0]$	$T(stack[0]) = T(stack'[0])$
8*	***GetField*** *index*	1	$stack[0] \leftarrow stack'[0].instanceField$	$T(stack[0]) = T(stack'[0].instanceField) \cup T(stack'[0])$
9	***GetStatic*** *index*	1	$stack[0] \leftarrow ClassName.staticField$	$T(stack[0]) = T(ClassName.staticField)$
10	***PutField*** *index*	1	$stack'[1].instanceField \leftarrow stack'[0]$	$T(stack'[1].instanceField) = T(stack'[0])$
11	***PutStatic*** *index*	1	$ClassName.staticField \leftarrow stack'[0]$	$T(ClassName.staticField) = T(stack'[0])$
12*	***Return(non-void)***	5	$callerStack[0] \leftarrow calleeStack'[0]$	$T(callerStack[0]) = T(calleeStack'[0])$
Index	**Detailed instruction semantics (The semantics of the instructions whose index are underlined serve as examples)**			
1	Push a constant value C onto the operand stack ($stack[0]$ represents the value at the stack top after an operation).			
2, 3	Load the value of the $\#index$ local variable onto the operand stack.			
4, 5	Pop and store the value at stack top to the $\#index$ local variable ($stack'[0]$ represents the value at the stack top before an operation).			
6, 7	Perform the binary operation $\otimes$ on the two values popped from the operand stack (i.e., $stack'[0]$ and $stack'[1]$) and push the result back onto the stack.			
8, 9	Get a field value of an object on the heap and push the value onto the operand stack. The object reference is popped from the stack (i.e., $stack'[0]$).The object field's name and type can be found by referring to the $\#index$ slot of the constant pool.			
10, 11	Pop and store the value at the stack top (i.e., $stack'[0]$) to an object field on the heap. The object reference is popped from the stack (i.e., $stack'[1]$). The object field's name and type can be found by referring to the $\#index$ slot of the constant pool.			
12	Pop the value at the callee's operand stack top (i.e., $calleeStack'[0]$), and push the value onto the caller's operand stack.			

```
1. public void onSensorChanged(SensorEvent event){
2.   if(event.sensor.getType() == ACCELEROMETER){
3.     boolean switchColor = isShuffled(event);
4.     if(switchColor){
5.       showMessage("Device shuffled");
6.       if(getBackgroundColor() == RED){
7.         setBackgroundColor(GREEN);
8.       } else{
9.         setBackgroundColor(RED);
10.      }
11.    }
12.  }
13. }
```

```
20. public boolean isShuffled(SensorEvent event){
21.   float[] values = event.values;
22.   float x = values[0];
23.   float y = values[1];
24.   float z = values[2];
25.   float g = SensorManager.GRAVITY_EARTH;
26.   float accelerationSquareRoot
27.         = (x * x + y * y + z * z) / (g * g);
28.   updateAccTextView(accelerationSquareRoot);
29.   if(accelerationSquareRoot >= 2){
30.       return true;
31.   }
32.   return false;
33. }
```

Figure 3.6: Example code to demonstrate taint propagation

(at the top of the operand stack after the calculation, represented by `stack[0]` in Table 3.6) would be tainted with the same marks if any operand (at the top of the operand stack before the calculation, represented by `stack'[0]` and `stack'[1]` in Table 3.6) is tainted before calculation. Other taint propagation rules can be explained similarly.[10]

We illustrate the taint propagation process by a concrete example. Figure 3.6 lists the code snippet from an application that uses accelerometer data to compute and display a phone's current acceleration status (Lines 21–28). The application also monitors whether the phone is being shuffled (Line 3), and if yes, it would change its background to a different color and notify its user (Lines 4–11). In this example, the initial taint mark is associated with an object reference `event`. The `event` object contains the sensory data from a smartphone's accelerometer. By object field access, the local array `values` of the `isShuffled` method get its assignment from the `event` object (Line 21). Since `values` is data dependent on the tainted object `event`, the taint mark is propagated to `values` according to Rule 8 (for handling object field reading instructions) and Rule 5 (for handling array element writing instructions). Then, by array element readings and local variable assignments, this taint

[10]Notes for Table 3.6: For Rule 8, we followed TaintDroid's choice to propagate object reference's taint to retrieved object field values to avoid undertainting in certain cases [48]. For example, we only taint the reference of sensory data objects (instead of tainting all object fields since the object can have complex structures) when taint propagation starts. Rule 8 can correctly help propagate taint marks when the sensory data object fields are read (see Figure 3.6 for illustration). Rule 12 does not conflict with the rule for handling control dependencies (see the "propagation taint marks" part in Section 3.2.4). They can be applied together.

mark is propagated to local variables `x`, `y`, and `z` (Lines 22–24) according to Rule 3 (for handling array element reading instructions) and Rule 4 (for handling local variable assignment instructions). Next, a local variable `accelerationSquareRoot` is calculated (Line 26–27). It is tainted according to Rule 6 (for handling binary calculation instructions) and Rule 4 since it is data dependent on the tainted local variables `x`, `y`, and `z`. Finally, method `isShuffled`'s return value is tainted according to a special rule that handles control dependencies. The rule taints a method's return value if any of its arguments is tainted (to be further explained shortly). Later this return value is further assigned to local variable `switchColor` in method `onSensorChanged` (Line 3), and `switchColor` is also tainted with the same mark (Rule 4). This completes the whole taint propagation process.

In our tainting process, we mainly consider data dependencies. Regarding control dependencies, we adopt a strategy similar to those studied in related work [41, 90]. That is, we taint a method's return value if any of its arguments is tainted (including the method's implicit "`this`" argument if applicable). This strategy/rule is based on the assumption that a method's output (i.e., return value) should depend on its input in well-written programs. This is the only rule concerning control dependencies in our taint propagation process. We do it this way because tracking finer-grained control dependencies may incur significant performance overhead and even imprecision to analysis results [48, 61]. Our taint propagation terminates when the application under analysis finishes its handling of sensor event.[11] This occurs in two situations. If the sensor event handler (e.g., `onSensorChanged()` in our example) does not start any worker thread to further handle the received sensor event, the propagation stops at the exit of this handler. Otherwise, the propagation has to continue until the sensor event handler returns and all worker threads terminate. Our taint propagation can thus identify the program data that depend on collected sensory data and trace their usages when an application executes. One thing that deserves explanation is that there might be cases where an application starts worker

[11] One can also track the usage of sensory data until an application exits or new sensory data arrive, but we did not observe any noticeable difference in our analysis results in experiments.

threads in a special way, e.g., these threads are delayed in their running, periodically started by a timer or kept long-running for handling sensor events. Although we did not observe similar cases in our study, there is no restriction of using such multi-threading features in Android applications. When such cases occur, our taint propagation would theoretically have to continue until all worker threads end. However, in practice, this may compromise the tool's usability since it can perform taint propagation for very long time and fail to report analysis results in a timely fashion. Therefore, for practicality, one may wish to set a timeout value for restricting such long taint propagation. This is an implementation issue and we do not elaborate further.

C. Analyzing sensory data utilization

With program data tainted with marks associated with sensory data, we can analyze how sensory data are used in an Android application and whether the uses are effective with respect to energy cost.

Consider an Android application's execution t_i, in which the application visits a set of states S_{t_i} by handling received events (user events, system events,[12] or sensory events), and finally terminates with all its components destroyed. As mentioned earlier, when we fix an upper bound b for the length of user interaction event sequences, the space of explored states S_b for this application would be bounded (i.e., the total number of states in this space is finite). As such, we are able to analyze these states to understand how sensory data are used, and compare their usages across different states. For comparison purposes, we propose an analysis metric called *Data Utilization Coefficient* (*DUC* for short). It is defined by Equation 3.1:

$$DUC(s,d) = \frac{usage(s,d)}{Max_{s' \in S_b, d' \in D}(usage(s',d'))} \tag{3.1}$$

[12]In GreenDroid, system events are generated by monitoring API invocations. For example, a broadcast message event will be generated when GreenDroid observes the invocation of the corresponding message broadcast API.

The utilization coefficient of sensory data d at state s is defined as the ratio between d's usage at state s and the maximal usage of any sensory data from our data pool D at any state in S_b. A lower DUC value indicates a lower utilization of sensory data. The usage of sensory data d at state s is further defined by Equation 3.2:

$$usage(s, d) = \sum_{i \in API\_Call(s,d)} eTest(i, d, s) \times noInst(i) \tag{3.2}$$

In this equation, $API\_Call(s, d)$ is the set of API call instructions executed since sensory data d are fed to the application at state s and until the data handling is finished. Function $eTest(i, d, s)$ is an effectiveness test to see whether the following two conditions both hold: (1) the API called by i uses program data dependent on sensory data d, and (2) the API's execution at state s produces perceptible benefits to users. When both conditions hold, the effectiveness test function returns 1. Otherwise, it returns 0. Function $noInst(i)$ returns the number of bytecode instructions executed by this API call. *The rationale behind our usage metric is that it reflects how many times and to what extent sensory data are used by an application at certain states to benefit its users.* This metric is designed based on our earlier study of 30 open-source Android applications that use sensors. These applications have called various Android or third-party APIs (e.g., Google Maps APIs) to use sensory data to support phone users with various functionalities.

Now we explain how the effectiveness test function $eTest(i, d, s)$ is implemented. For its first condition, we check whether the concerned API is called with arguments (including its implicit "`this`" argument if applicable) having the same taint mark as sensory data d. For its second condition, we take an outcome-based strategy. The basic idea is that the API called by instruction i at state s passes the effectiveness test if and only if its execution produces perceptible outcomes/benefits to users (e.g., updating visible GUIs or writing to file systems). Specifically, our current strategy works as follows:

- If the API updates GUI elements, it passes the test as long as these GUI elements are visible at application state s, and fails otherwise.

Table 3.7: GPS data utilization coefficients at three application states of Osmdroid

Application state	Method calls that consume GPS data	GPS data usage	GPS data utilization coefficient
$\langle s_0, [e_1, e_2] \rangle$	*processLocation, putExtra, sendBroadcast**, *getExtra, updateMap**, *persistToDatabase**	$3n$	$3n$ / $3n = 1.00$
$\langle s_0, [e_1, e_3] \rangle$	*processLocation, putExtra, sendBroadcast**, *getExtra, updateMap*†	n	n / $3n = 0.33$
$\langle s_0, [e_1, e_2, e_3] \rangle$	*processLocation, putExtra, sendBroadcast**, *getExtra, updateMap*†, *persistToDatabase**	$2n$	$2n$ / $3n = 0.66$

- e_1: users start Osmdroid and the map activity launches; e_2: users switch on the location tracking mode; e_3: users switch from map activity to another activity.
- Method calls that can pass the effectiveness test are marked with the symbol "*"; method calls used to update invisible GUI elements are marked with the symbol "†".
- Please note that only method calls marked with the symbol "*" use GPS data and produce perceptible benefits to Osmdroid users.

- If the API: (1) stores any data to file systems, databases or network, (2) updates a phone's status (e.g., adjusting its screen brightness), or (3) passes any message for inter- or intra-application communication (e.g., broadcasting system-wide events), the API passes the test regardless of the application state. Here, we conservatively assume that the stored data or passed messages will eventually produce perceptible benefits to users.

- For all other cases, the API fails the test.

As such, our analysis can identify those application states where sensory data are underutilized based on calculated sensory data usage and cross-state comparisons. We give one example for illustration. Consider the three states in the Osmdroid example in Figure 3.3(b). They are also listed in Table 3.7. Take the third state $\langle s_0, [e_1, e_2, e_3] \rangle$ for example. It means that: Osmdroid's user starts the application by launching `MapActivity` (e_1), enables its location tracking functionality (e_2), and switches the application to another activity (e_3). We analyze sensory data utilization for these three states. For ease of presentation, we explain at a source code level (actual analysis is conducted at a bytecode instruction level), and assume that: (1) each method is a pre-defined API, and (2) there are n bytecode instructions executed

==========

Sensory Data Underutilization

==========

[Sensory data usage]: *sendBroadcast*, *updateMap*†

[Sensory data utilization coefficient:] 0.33

[Event handler calling trace]:

MapActivity.onCreate (Line 4), *MapActivity.onStart*, *MapActivity.onResume*, *GPSService.onCreate* (Line 34), *MapActivity.onPause*, *MapActivity.onStop*, *gpsListener.onLocationChanged* (Line 38), *myReceiver.onReceive* (Line 8)

Notes: (1) "†" highlights APIs that ineffectively utilize sensory data. (2) For ease of understanding, we use class, variable and handler names to represent event handlers, while in real reports the event handlers are represented using object IDs and fully qualified Java method signatures. (3) Our tool will also output source file names and source line numbers if they are available.

Figure 3.7: Example analysis report of GreenDroid

for each called API. Consider the second state, which is reached when the user switches to another activity from `MapActivity` directly. For this state, the location tracking functionality is not yet enabled. We observe that all external GPS data and internal program data depending on these GPS data are processed and used in turn by a set of APIs, namely, `processLocation`, `putExtra`, `sendBroadcast`, `getExtra` and `updateMap`. According to our usage metric, only the `sendBroadcast` API passes the effectiveness test. The other four APIs fail the test because none of them can produce perceptible benefits to users (note that the map is still invisible now). According to Equation 3.2, the GPS data usage at this state is n. We can also calculate that GPS data would have a maximal usage of $3n$ at the first state, where `updateMap` is used to render a visible map, `sendBroadcast` spreads the GPS data to the entire system, and `persistToDatabase` method stores the GPS data to database. Therefore, the GPS data utilization coefficient for the second state is 0.33 ($= n \ / \ 3n$). The coefficients for the other two states can be calculated similarly, as shown in Table 3.7. These results suggest that GPS data are clearly underutilized at the second state, as compared to the other two states.

Our GreenDroid implementation ranks sensory data utilization coefficients for different application states such that energy problem reports can be prioritized and developers can then focus on the most serious energy problems. These reports contain two major pieces of information to ease energy problem diagnosis and fixing.

First, GreenDroid reports how sensory data are consumed by different APIs at different application states, and highlights those APIs that ineffectively use sensory data. Second, GreenDroid provides concrete event handler calling traces (corresponding to user interaction event sequences). For ease of understanding, we give an example report in Figure 3.7. It shows that GPS data are not well-utilized by Osmdroid at the second application state described in Table 3.7. In this example, GreenDroid reports that: (1) GPS data are used to render an invisible map (i.e., `updateMap` API invocation), and (2) an event handler calling trace to reach the problematic application state. Such reported information are actionable to developers. By examining reported event handler calling traces, developers will be able to construct concrete test cases (e.g., user interaction events) to reproduce the corresponding sensory data underutilization scenario. For instance, the event handler calling trace in our example report corresponds to the following two user interaction events: (1) launching the `MapActivity`, and (2) switching away from `MapActivty` (see Section 2.1 for the calling order of activity lifecycle event handlers). Besides, by examining reported sensory data usages, especially ineffective data usages (e.g., `updateMap` in this example), developers can understand why an application consumes more energy than necessary. Such energy problem reports provide much richer information than pure complaints that can be commonly found in smartphone application forums [82]. Developers can thus pinpoint those problematic application states where energy is consumed unnecessarily due to ineffective use of sensory data. They can then take various actions for problem fixing, e.g., tuning down sensing rates or temporally disabling sensing as discussed in our earlier examples.

Finally, for detected missing sensor or wake lock deactivations, GreenDroid will also report similar information for energy problem diagnosis. Specifically, it will report: (1) those sensor listeners or wake locks that are forgotten to be properly unregistered or released before an application exits, and (2) event handler calling traces for reaching those problematic application states.

3.3 Experimental Evaluation

We implemented our energy diagnosis approach as a prototype tool named GreenDroid [22] on top of JPF [93]. GreenDroid consists of 18,367 lines of Java code, including 7,251 lines of code for energy diagnosis, and other 11,116 lines of code for modeling Android APIs. We explain some details about GreenDroid's implementation. First, modeling Android APIs is necessary for our diagnosis because Android applications depend on a proprietary set of library classes that are not available outside real devices or emulators [73]. These library classes are mostly built on native code. Due to JPF's closed-world assumption [93], we have to model these library classes and their exposed APIs. Ignoring this modeling requirement would result in imprecision in the diagnosis results. For example, if GreenDroid does not properly model the Activity class's `startActivity()` API, it will not be able to analyze activity switches, which are very common in Android applications. However, Android exposes more than 8,000 public APIs to developers [50]. Fully modeling them is extremely labor-intensive and almost impossible for individual researchers like us. As such, in our current implementation, we took a pragmatic approach by manually modeling a subset of APIs that are commonly called in Android applications. Modeling these APIs is already sufficient for carrying out our evaluation with real application subjects. To be specific, we have carefully modeled 76 APIs using JPF's native peer and listener mechanisms [22, 64]. These APIs either frequently get invoked in our experimental application subjects or have to be modeled as otherwise JPF will crash on their invocation (e.g., when they involve native calls). Modeling these APIs took us nearly three months. For remaining APIs, we provided stubs with simple logics. In these stubs, we basically ignored their corresponding APIs' side effect if any, and made them return a value selected from a reasonably bounded domain when necessary. Second, besides tracking standard JPF program state information (e.g., call stack of each thread, heap and scheduling information) [86], GreenDroid also tracks the following four types of information for analysis: (1) a stack of active activities, their lifecycle status, and visibility of their containing GUI elements, (2) a list of running services and their lifecycle status, (3) a list of reg-

istered broadcast receivers, and (4) a list of registered sensor listeners and wake locks. More tool implementation details can be found in our technical report [64] and research paper [69].

In this section, we evaluate GreenDroid by controlled experiments and a case study. We aim to answer the following five research questions:

- **RQ4 (Effectiveness and efficiency):** *Can GreenDroid effectively diagnose and detect energy problems in real-world Android applications? What is its diagnosis overhead?*
- **RQ5 (Necessity and usefulness of AEM model):** *Can GreenDroid correctly schedule event handlers for Android applications with our AEM model? Can GreenDroid still conduct an effective diagnosis if it randomly schedules event handlers (i.e., with our AEM model disabled)?*
- **RQ6 (Impact of event sequence length limit):** *How does the length limit of generated user interaction event sequences affect the thoroughness of our energy diagnosis in terms of code coverage?*
- **RQ7 (Comparison with resource leak detection work):** *How is GreenDroid compared with existing resource leak detection work in terms of finding real missing sensor or wake lock deactivation problems?*
- **RQ8 (Energy saving):** *How much energy can be potentially saved if our detected energy problems are fixed?*

3.3.1 Experimental Setup

We selected 14 open-source Android applications as our experimental subjects. Table 3.8 lists their basic information, which includes: (1) version number, (2) size of the selected version, (3) repository from which source code was obtained, (4) application category, and (5) number of downloads.[13] The first 12 applications were

[13]The number of application downloads reported here may slightly differ from what was reported in our empirical study due to the data update during the time gap between our empirical study and experiments.

Table 3.8: Experimental subject information and detected energy bugs

Application Name	Version	LOC	Source code availability	Category	Downloads	Detected energy problem (severity level)
DroidAR	R-204[1]	18,106	Google Code	Tools	5K ~ 10K	Missing sensor deactivation (Medium[3])
Recycle Locator	R-68	3,241	Google Code	Travel & Local	1K ~ 5K	Missing sensor deactivation (Medium)
Ushahidi	R-9d0aa75	10,186	GitHub	Communication	10K ~ 50K	Missing sensor deactivation (N/A)
AndTweet	V-0.2.4[2]	8,908	Google Code	Social	10K ~ 50K	Missing wake lock deactivation (Medium)
Ebookdroid	R-137	14,351	Google Code	Productivity	1M ~ 5M	Missing wake lock deactivation (Medium)
BableSink	R-12879a3	1,718	GitHub	Library & Demo	1K ~ 5K	Missing wake lock deactivation (N/A)
CWAC-Wakeful	R-d984b89	896	GitHub	Education	1K ~ 5K	Missing wake lock deactivation (N/A)
Sofia Public Transport Nav.	R-114	1,443	Google Code	Transportation	10K ~ 50K	Sensory data underutilization (Critical)
	R-115	1,427	Google Code	Transportation	10K ~ 50K	Missing sensor deactivation (Critical)
Osmdroid	R-750	18,091	Google Code	Travel & Local	10K ~ 50K	Sensory data underutilization (Medium)
Zmanim	R-322	4,893	Google Code	Books & References	10K ~ 50K	Sensory data underutilization (Critical)
Geohash Droid	V-0.8.1-pre2	6,682	Google Code	Entertainment	10K ~ 50K	Sensory data underutilization (Medium)
My Tracks	R-7749d47	16,560	Google Code	Health & Fitness	5M ~ 10M	Sensory data underutilization (N/A)
Omnidroid	R-863	12,427	Google Code	Productivity	1K ~ 5K	Sensory data underutilization (Critical)
GPSLogger	R-15	659	Google Code	Travel & Local	1K ~ 5K	Sensory data underutilization (Medium)

[1,2]: Symbol "R" stands for "revision" and symbol "V" stands for "version";

[3]: We obtained the problem severities from corresponding applications' bug tracking systems. "N/A" means that developers did not explicitly label problem severities.

confirmed to have energy problems of our two identified patterns (Section 3.3). We use them to validate the effectiveness of our approach. We also selected two other subjects (Omnidroid and GPSLogger) from the open-source applications collected in our empirical study. Neither of these two applications have confirmed energy problem reports. However, from their project websites and user reviews, we judged that they heavily use GPS sensors in a very energy-consuming way and are susceptible to energy inefficiency problems. Thus we also selected them for our study to see whether our approach can identify energy optimization opportunities for them. We observe from Table 3.8 that our selected applications have been popularly downloaded (over five million downloads in total), and covered a variety of application categories (11 different categories). We obtained these applications' source code and compiled them on Android 2.3.3 for our experiments.[14] We conducted our experiments on a dual-core machine with Intel Core i5 CPU @2.60GHz and 8GB RAM, running Windows 7 Professional SP1. In the following we elaborate on our experiments with respect to the five research questions in turn.

3.3.2 Effectiveness and Efficiency of our Approach

To answer research question RQ4 about GreenDroid's effectiveness and efficiency, we ran GreenDroid to diagnose each application listed in Table 3.8 and recorded its diagnosis overhead. In this set of experiments, we controlled GreenDroid to generate sequences of at most six user interaction events for each application execution (not including the first events for "launching entry activity" when our analysis starts and the last events for "finishing active activities and services" when our analysis ends). This is for cost-effectiveness and it already enabled GreenDroid to explore quite a large number of application states to expose energy problems as we will show later. We examined top ranked diagnosis reports, especially those with highlighted

[14] At our study time, we chose Android 2.3.3 because it was one of the most widely adopted Android platforms and is compatible with most applications on the market. Our approach is general and not restrictive to specific platform versions though.

ineffective API calls, to see whether they can locate real energy problems in these applications.

We observed that GreenDroid successfully located 15 real energy problems in these applications, as listed in Table 3.8. Four of them are caused by missing sensor deactivation, four by missing wake lock deactivation, and the remaining seven by sensory data underutilization. The first 13 energy problems listed in Table 3.8 have been confirmed by developers prior to our experiments. In addition, GreenDroid successfully found two potential energy problems in Omnidroid and GPSLogger. These two problems were previously unknown. We submitted our bug reports to corresponding developers, and they were both confirmed. GPSLogger developers even invited us to join their team to help improve GPSLogger's energy efficiency. Besides, as shown in Table 3.8, the severity levels of our detected 15 problems range from "medium" to "critical". This indicates that such problems can cause serious energy waste. Indeed, we found many negative comments complaining about battery drain issues from the bug tracking systems and Google Play store user review pages of the concerned applications (e.g., Geohash Droid, AndTweet and Zmanim). We discuss some of these energy problems in detail below.

A. Missing Sensor or Wake Lock Deactivation

Android API documentation recommends developers to unregister sensor listeners and release wake locks when they are no longer needed [1]. However, we found that missing sensor or wake lock deactivation is common in Android applications. GreenDroid detected eight applications suffering such energy problems from our 14 subjects. These problems happened because developers either forgot to unregister sensor listeners or release wake locks, or performed these operations incorrectly. For example, the code snippets in Figure 3.8 demonstrate how Ushahidi developers wrongly unregistered a GPS listener. We observe in the buggy version that, developers registered a GPS listener `gpsListener` in the `onCreate()` handler of the `CheckInMap` activity (Lines 3–6), and then tried to unregister the listener in the `onDestroy()` handler of `CheckInMap` (Lines 10–11). However, instead of passing previously registered `gpsListener` to the sensor listener unregistration API `removeUpdate()`,

```
/**buggy version of the CheckInMap class**/
1. public class CheckinMap extends MapActivity {
2.   public void onCreate(){
3.     MyGPSListener gpsListener = new MyGPSListener();
4.     LocationManager lm = getSystemService(LOCATION_SERVICE);
5.     //GPS listener registration
6.     lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
7.   }
8.   public void onDestroy() {
9.     //unregister GPS listener
10.    getSystemService(LOCATION_SERVICE)
11.           .removeUpdates(new MyGPSListener());
12.  }
13.  //location listener class
14.  public class MyGPSListener implements LocationListener {
15.    public void onLocationChanged(Location loc) {
16.      //utilize location data
17.    }
18.  }
19. }

/**correct version of the CheckInMap class**/
20. public class CheckinMap extends MapActivity {
21.   private MyGPSListener gpsListener;
22.   private LocationManager lm;
22.   public void onCreate(){
23.     gpsListener = new MyGPSListener();
24.     lm = getSystemService(LOCATION_SERVICE);
25.     //GPS listener registration
26.     lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
27.   }
28.   public void onDestroy() {
29.     //unregister GPS listener
30.    lm.removeUpdates(gpsListener);
31.   }
32. }
```

Figure 3.8: The energy bug in Ushahidi application

developers wrongly created a new GPS listener instance and passed its reference to `removeUpdate()`. The consequence is that the previously registered sensor listener `gpsListener` was not properly unregistered.

For performance considerations, the Android OS keeps an application process alive as long as possible, until the system runs low on resources (e.g., memory). According to this policy, even a dummy process that hosts no application component is not guaranteed to be terminated in a timely fashion [1]. Therefore, in the buggy version, the `gpsListener` instance would remain in memory for a long time even if the activity it belongs to has been destroyed. The activity instance could also remain in memory after its `onDestroy()` handler is called. As a result, valuable battery energy can be wasted by unnecessary GPS sensing. Ushahidi's developers later realized this problem from bug reports and fixed it. Figure 3.8 also gives the correct version for comparison.

B. Sensory data underutilization

GreenDroid also detected seven applications suffering from sensory data underutilization problems out of our 14 subjects. Among these detected problems, three (Table 3.8) are critical ones that can cause massive energy waste. We discuss these seven problems in detail below.

Osmdroid. Osmdroid is a navigation application similar to Google Maps. After diagnosis, GreenDroid reported that Osmdroid's location data utilization coefficient is no more than 0.2239 for 30.51% explored states, but close to 1 for other states, as shown in Figure 3.9(a).[15] This strongly suggests that Osmdroid poorly utilizes location data at certain states. We examined the reports generated by GreenDroid and quickly found that if users switch from `MapActivity` to other activities without

[15]Notes of the Figure 3.9: (1) In the figures, the location utilization coefficient is accurate to four decimal places. (2) Two states with indistinguishable utilization coefficients (i.e., cannot be distinguished by four decimal places) are shown in the same bar. (3) Utilization coefficients with very few occurrences (i.e., less than 5) are not shown in the figures for ease of presentation, so the percentages in each figure may not add up to 100%. (4) The total number of states for each application does not equal the number of explored states reported later because the location sensing is not enabled in some explored states.

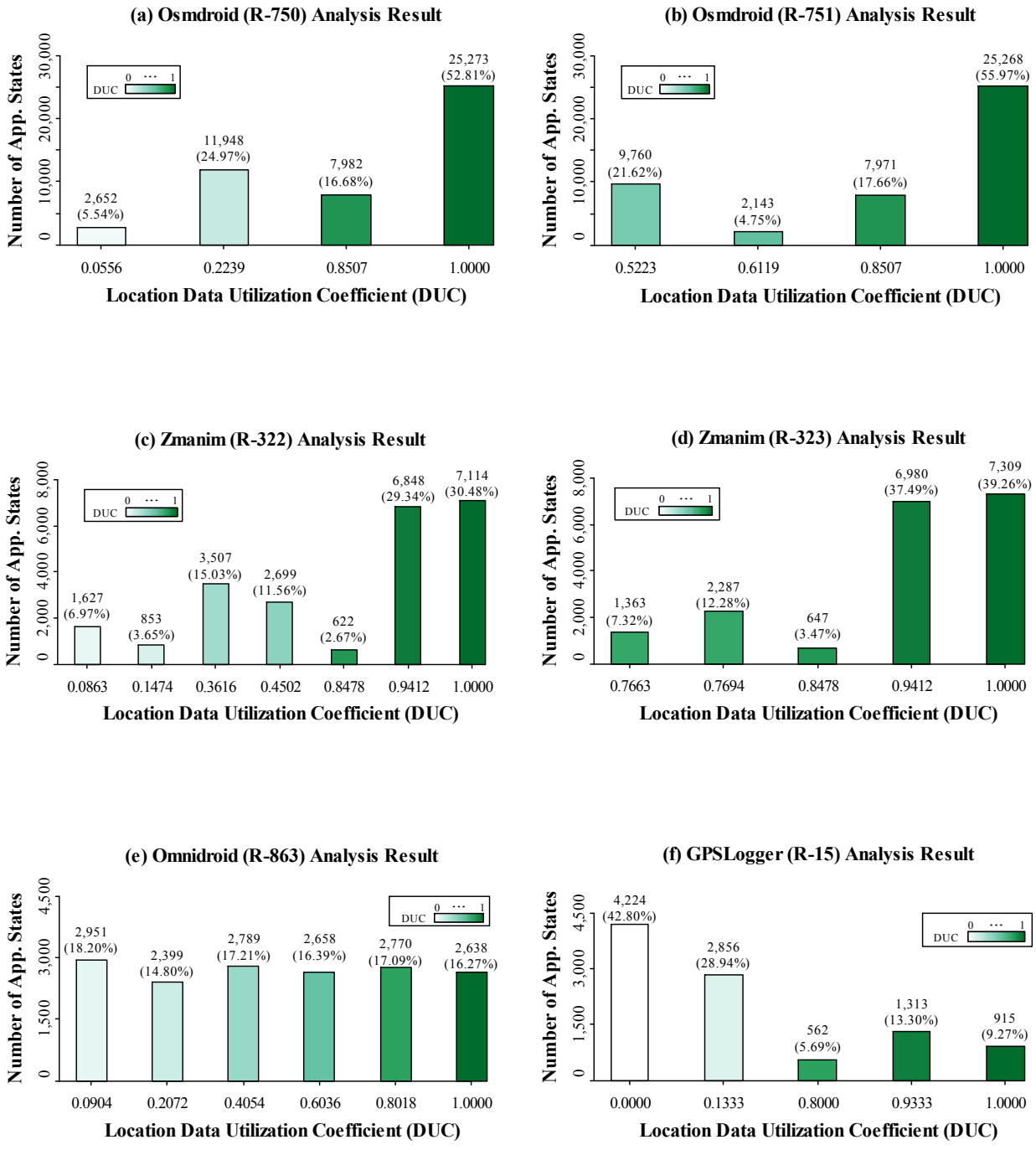

Figure 3.9: Sensory data utilization analysis results (part 1)

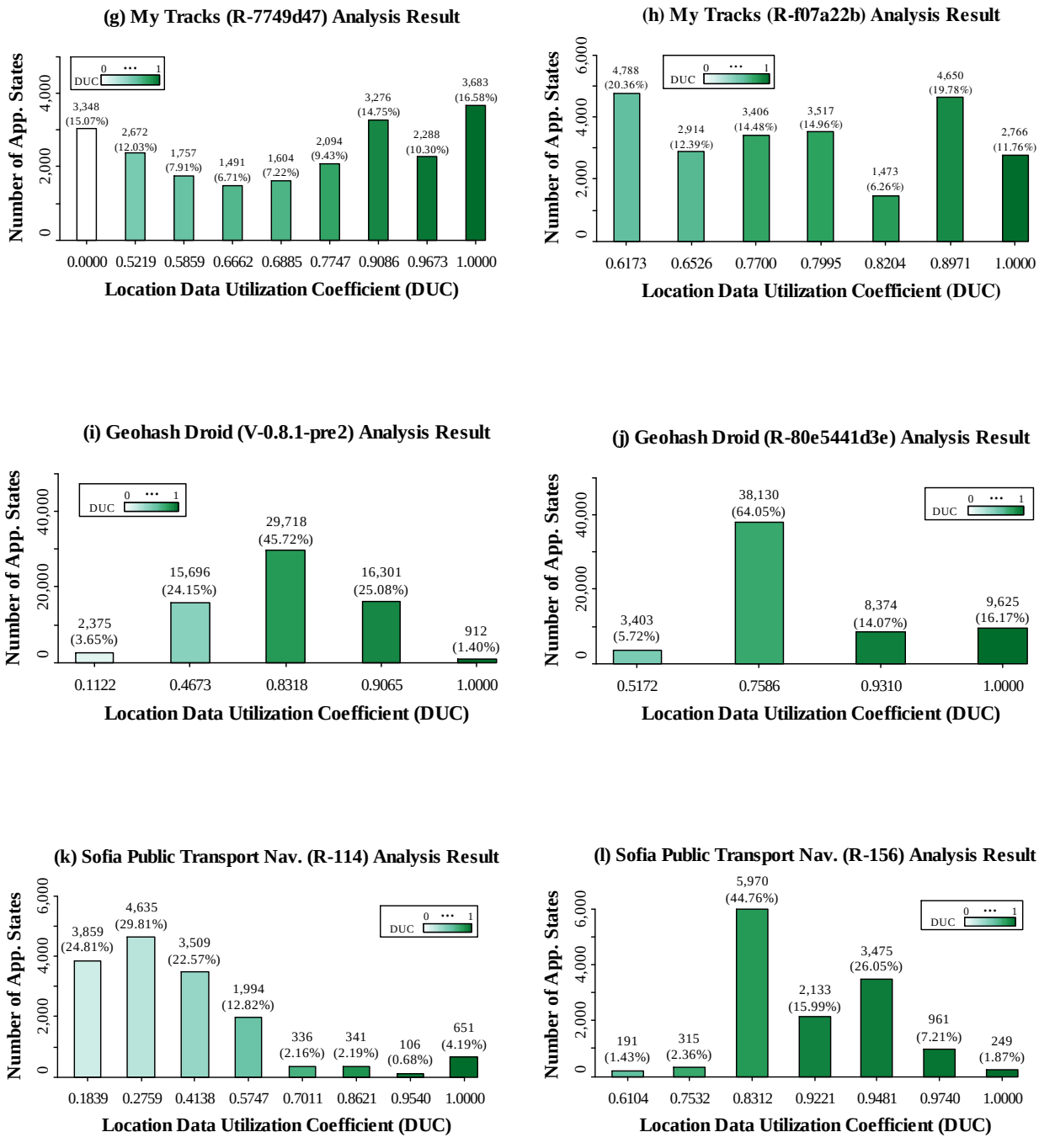

Figure 3.9: Sensory data utilization analysis results (part 2)

enabling location tracking, location data would be used to render an invisible map (recall that GreenDroid can highlight ineffective API calls). This greatly wastes valuable battery energy as reported by users [28] (issue 53). To fix this problem, developers later disabled GPS sensing if users leave `MapActivity` without the location tracking functionality enabled. Figure 3.9(b) gives the new version's location data utilization analysis result. We can observe that location data are now much better utilized with a utilization coefficient above 0.5223.

Zmanim. Zmanim is a location-aware application for reminding Jewish people about prayer time during the day (i.e., zmanim). The application generates zmanim according to users' locations and corresponding time zones. Interestingly, developers already realized that location sensing could be energy-consuming, and they made the application stop location sensing once its required locations are obtained. However, as Figure 3.9(c) shows, GreenDroid still reported that for 37.37% explored states, Zmanim's location data utilization coefficient is no more than 0.4502, but close to 1 for other states. This energy problem is similar to what we found in Osmdroid. If users switch from the location sensing activity to other activities before the required locations are successfully obtained, battery energy would keep being wasted to update invisible GUI elements. In scenarios where GPS signals are rather weak, users frequently complained that Zmanim caused huge battery drain [37]. We give an example of such complaints below. Similar to Osmdroid, Zmanim developers also later disabled location sensing in such problematic cases, and we give the new version's location data utilization analysis result in Figure 3.9(d) for comparison (much improved utilization).

> **Zmanim Issue 56:** *"I should see GPS icon only until a location is obtained. After that, GPS should be turned off. However, even if turning off GPS once a fix is obtained, this issue remains as a bug, since a user could hit home button before the fix is obtained, therefore leaving GPS on. These bugs quickly kill my battery."*

Omnidroid. Omnidroid helps automate system functionalities based on user contexts. For example, Omnidroid can help users automatically send a reply mes-

sage such as "busy in a meeting" when they receive a phone call during an important meeting. When Omnidroid runs, it maintains a background service to periodically check location updates. If any location update satisfies a pre-specified condition, its corresponding action would be executed as a response. Our diagnosis results in Figure 3.9(e) show that 18.2% explored states have a location data utilization coefficient of no more than 0.0904. We found that at these states, users have not specified any condition or chosen any action. In other words, location data are collected for no use except being stored to a database for logging purposes (this explains why the location data utilization coefficient is not 0). Then why does this background service keep collecting location data? It could cause huge energy waste. We reported this problem (previously unknown) to Omnidroid developers, and suggested enabling location sensing only when there are conditions/rules concerning user locations. We then received a prompt confirmation and developers marked our reported problem as "critical" [27]:

> **Omnidroid Issue 179:** *"Completely true, and your suggestion is a great idea and you're correct Omnidroid does suck up way more energy than necessary as a result. I'd be happy to accept a patch in this regard."*

GPSLogger. GPSLogger collects users' GPS coordinates to help them tag photos or visualize their traces. Figure 3.9(f) presents our diagnosis results for its GPS data utilization. We found that for 42.80% explored states, GPS data have not even been utilized. The utilization coefficient is 0. For the next 28.94% states, the coefficient is also low at 0.1333, while for other states, it is close to 1. We examined the diagnosis reports and found another new energy problem that has not yet been reported. Similar to Omnidroid, GPSLogger also maintains a background service to collect GPS data. It continually evaluates whether collected GPS data satisfy certain precision requirements. If yes, the data are processed and stored to a database, and GPSLogger would then update its GUI to notify users. Otherwise, the data are discarded. However, when GPS signals are weak, GPS sensors may keep collecting noisy data. These data mostly do not satisfy precision requirements and are actually discarded. This produces no benefits to users, and explains why

GPS data have a very low utilization coefficient at some states. This problem can be common when users enter an area where the GPS reception is bad. We submitted a bug report [21] (issue 7) to suggest temporarily slowing down or disabling location sensing when the application continuously finds its collected GPS data of low quality. Our bug report was confirmed by GPSLogger developers. They also invited us to help improve GPSLogger's energy efficiency [21]. We will further discuss our patch later in Section 3.3.6.

My Tracks. My Tracks collects GPS data for recording users' path, speed, distance and elevation while they do outdoor exercises (e.g., running, cycling). We diagnosed the GPS data utilization of My Tracks using GreenDroid. Our tool reported that My Tracks does not utilize GPS data for 15.07% explored states, as shown in Figure 3.9(g). In other words, the battery energy spent on GPS sensing at these application states is completely wasted. We checked GreenDroid's diagnosis reports and the source code of the application and confirmed the problem. Similar to GPSLogger, at these problematic states, My Tracks simply discards any collected GPS data. The energy waste occurred due to a mistake in the application's implementation. My Tracks uses a long running background service to handle GPS updates. The service registers a location listener right upon its launching and will not unregister this listener until it is destroyed. However, the location listening is in fact only necessary after users start the recording and before they stop the recording. Then in those application states when there is no recording initiated by users but the service is actively running, the obtained location data will not be processed and stored (discarded instead), leading to energy waste. Developers later realized this problem and fixed it eventually by registering location listener after users start recording, and unregistering the listener after the recording is stopped [25] (revision f07a22b). For comparison, we further analyzed the GPS location utilization of the application after developers fixed the problem. As Figure 3.9(h) shows, there are no application states where GPS data are not utilized in the fixed version.

Geohash Droid. Geohash Droid is an entertainment application for adventure enthusiasts. It randomly picks up a location for adventure, and navigates its users to that location using GPS data. We diagnosed Geohash Droid and found that its

utilization coefficient is no more than 0.4673 for 27.80% explored states, as shown in Figure 3.9(i). We studied the diagnosis reports and found that at these states, GPS data were used only to show the users' current locations in an icon on the phone's notification bar (a phone's notification bar is a GUI element container that is outside an application's normal GUI and is always visible when the phone's screen is on). However, in other states, GPS data were also used to update the navigation map as well as computing detailed travel information (e.g., distance to destination). This comparison shows that GPS data were not well utilized in those 27.80% explored states, and this could cause energy waste. After realizing this, Geohash Droid's developers made a patch to slow down the application's GPS sensing rate to every 30 seconds to save energy when GPS data are only used for updating the notification bar [16] (issue 24). Figure 3.2 shows their comment after patching, and both their own testing and user feedbacks confirmed that there is indeed a significant improvement in Geohash Droid's energy efficiency [16]. Besides, in later revisions to Geohash Droid, developers redesigned the application by completely removing this notification icon. They chose to automatically switch off GPS updates when the navigation map and detailed information screen become invisible (see revision 80e5441d3e for details). We analyzed this new version and present the result in Figure 3.9(j) for comparison. The result shows that in 94.29% explored states, the GPS data are now effectively utilized.

Sofia Public Transport Nav. Sofia Public Transport Nav. uses its collected GPS data to locate the nearest bus stops for its users, and provides arrival time estimation for concerned buses by querying a remote server. GreenDroid diagnosed its GPS data utilization, and reported that GPS data were poorly utilized for 24.81% explored states, and for the next 52.38% states, the utilization coefficient was also below 0.4138, as shown in Figure 3.9(k). We examined the diagnosis reports and confirmed this energy problem. In Sofia Public Transport Nav., GPS data are mainly used to update a map that shows nearby bus stops. However, for many states, the

dialog box showing bus arrival time is at foreground,[16] hiding the map that shows nearby bus stops. Then because users may keep refreshing the dialog box to check bus arrival time, GPS data during this period will be used mainly to update the map hidden by the dialog. This is a waste of energy. The application developers later found this problem, and disabled its GPS update for states where the bus arrival time estimation dialog is at foreground. Interestingly, although developers closed the corresponding bug report [32] (issue 38) soon after creating this patch, they mistakenly introduced another missing sensor deactivation problem. In later development and communications with users, they realized this new problem and eventually fixed it [32] (issue 76). This story suggests that: (1) developers lack easy-to-use and effective tools to help detect energy problems in their applications, and (2) fixing sensory data underutilization problems is non-trivial and may instead introduce new energy problems. For comparison, we also analyzed the application after developers eventually fixed all energy problems including this new one. As the result in Figure 3.9(l) shows, there are now no application states whose GPS data utilization coefficient is significantly lower than others.

From the above discussions, we can see how automated sensory data utilization analysis can help diagnose energy problems for Android applications. When developers find that sensory data are clearly underutilized at certain states of their applications, they can consider whether their applications can reach these problematic states frequently and stay there for long time (e.g., an activity can be left to background until users explicitly switch back to it). If yes, developers may have to tune down the concerned sensors' sensing rates or even disable them, as otherwise energy cost can be very high, but produced benefits can be marginal instead. Besides, we also find that in large-scale application subjects like Omnidroid and Zmanim, their sensory data usage is very complex, involving hundreds of method/API calls. In such subjects, manually examining how sensory data are utilized can be extremely labor-intensive and error-prone. This justifies the great need for an automated di-

[16]GreenDroid models pop-up windows like dialog boxes by this strategy: (1) If a pop-up window is being displayed, GreenDroid considers all GUI elements underneath invisible; (2) If a pop-up window is dismissed, GreenDroid considers the GUI elements underneath visible again.

Table 3.9: GreenDroid diagnosis overhead and random execution result

Application name	Diagnosis information and overhead				Random event handler scheduling results (runtime exceptions)
	Explored states	Avg. number of handlers executed during each application execution	Diagnosis time (seconds)	Memory consumption (MB)	
DroidAR	91,170	60	284	233	67/100
Recycle Locator	114,709	44	46	162	4/100
Ushahidi	55,269	75	32	175	58/100
AndTweet	98,410	33	47	192	82/100
Ebookdroid	57,330	42	22	149	86/100
BableSink	42,987	63	15	154	17/100
CWAC-Wakeful	30,705	46	11	118	11/100
Sofia Public Transport Nav.	57,316	50	17	204	62/100
Osmdroid	120,189	43	159	575	79/100
Zmanim	54,270	34	114	237	31/100
Geohash Droid	144,710	60	185	229	71/100
My Tracks	82,137	45	207	341	74/100
Omnidroid	52,805	78	242	396	22/100
GPSLogger	58,824	28	41	153	9/100

agnosis tool like our GreenDroid to help locate potential energy problems caused by sensory data underutilization. To reduce developers' efforts in reading diagnosis reports, GreenDroid prioritizes these reports according to their sensory data utilization coefficients, and highlights ineffective API calls (e.g., those for updating invisible GUIs). This can help developers quickly figure out the causes of some subtle energy wastes.

C. Analysis overhead

Table 3.9 presents GreenDroid's diagnosis overhead. For each of our 14 subjects, it reports: (1) the number of application states GreenDroid explored, (2) the average number of event handlers GreenDroid executed during each application execution,

including those handlers for system events,[17] (3) diagnosis time, and (4) the amount of memory GreenDroid consumed. For each subject, we conducted experiments three times to obtain these results. The number of application states explored and event handlers executed in different runs remained the same. The diagnosis time and memory consumption slightly varied in different runs and Table 3.9 reports the averaged results.

We observe that GreenDroid could quickly explore thousands of application states and perform energy inefficiency diagnosis. For example, for the two largest subjects My Tracks (over 16K LOC) and DroidAR (over 18K LOC), GreenDroid explored over 80K states during its diagnosis and executed over 40 event handlers in each application execution (recall that GreenDroid executes each subject many times). It finished diagnosis within five minutes. The memory cost was less than 350 MB. Such overhead can be well supported by modern PCs, and compares favorably with state-of-the-art testing or debugging techniques, which typically take hours to explore up to 100K states [30]. This suggests that GreenDroid is a practical tool for diagnosing energy problems in real-world Android applications.

3.3.3 Necessity and Usefulness of AEM Model

To answer research question RQ5 about the usefulness of our proposed AEM model, we conducted two comparison experiments. First, we ran GreenDroid to diagnose our experimental subjects with the AEM model disabled, assuming that event handlers can be randomly scheduled. We examined whether GreenDroid could still locate energy problems in such a setting. Second, to study whether the executions of our experimental subjects in GreenDroid (with AEM model enabled) resemble real executions, we instrumented all 149 event handlers defined in our largest subject DroidAR, and conducted the following experiment. We randomly selected 50 execution traces of DroidAR generated by GreenDroid. These executions on av-

[17]System events could result in several consecutive handler calls. For example, an activity-destroying event may trigger the concerned activity's `onPause()`, `onStop()`, and `onDestroy()` handlers in turn.

erage involve 54 event handler calls (not necessarily distinct). We extracted from them corresponding user interaction event sequences. We then ran DroidAR in the Android emulator [1], which is included in the Android Software Development Kit, and manually provided the same user interactions (i.e., the same event sequences). We logged real event handler calling traces, and compared them with those from GreenDroid. We discuss the results of these experiments below.

First experiment. We observe that without AEM model (i.e., scheduling event handlers randomly), GreenDroid (actually JPF) already encountered great challenges in executing Android applications, not to mention diagnosing any of their energy problems. The last column of Table 3.9 lists these execution results. Among 100 application executions, we observed many runtime exceptions. For example, 79 out of 100 executions of Osmdroid failed because of runtime exceptions, and these exceptions also crashed JPF. We manually studied these exceptions, and found that most of them arose from ignoring data flow dependencies between event handlers. For instance, it is quite often that developers initialize a GUI widget instance in an activity's `onCreate()` handler, and later use this instance in other handlers. In random handler scheduling, if other handlers are wrongly scheduled before `onCreate()`, a null pointer exception may be thrown. Such exceptions cannot be easily addressed, and can cause termination of our energy diagnosis. For two small-sized subjects Recycle-locator and GPSLogger, fewer exceptions (4 and 9) were observed since their data flow dependencies between event handlers are relatively simple. Still, these exceptions seriously prevented GreenDroid from diagnosing our experimental subjects. Besides, even for cases where no exceptions occurred, we found that the diagnosis reports contain many meaningless handler calling traces that offer little information to help developers pinpoint energy problems. This suggests that our AEM model is indeed necessary for an effective diagnosis of energy problems in Android applications. In addition, since our AEM model is essentially an abstraction of event handler scheduling policies for the Android platform, it can easily be adapted and used in other analysis techniques for Android applications.

Second experiment. We observe that in 39 out of 50 executions, GreenDroid generated exactly the same handler calling traces as real executions. In the re-

maining 11 cases, GreenDroid failed to schedule event handlers in the same way as real executions did due to two major reasons. First, we did not consider dynamic GUI updates when implementing GreenDroid. This could make GreenDroid generate some user interaction events that are impossible in an Android emulator (and also in real devices), because they are invalid due to runtime GUI updates (4 cases). Second, GreenDroid did not model concurrency adequately in its current implementation because JPF did not fully model Java concurrency programming constructs (e.g., `java.util.concurrent.Executor` was not modeled). This caused GreenDroid to fail to handle some system events (e.g., broadcast events) that were triggered in some worker threads (7 cases). Although these two problems did not cause noticeable consequences on the effectiveness of our diagnosis, we will still consider addressing them in future releases of our GreenDroid. This requires non-trivial engineering effort.

3.3.4 Impact of Event Sequence Length Limit

Our research question RQ6 studies how the thoroughness of our energy diagnosis can be affected by the length limits on generated user interaction event sequences. To answer this question, we applied GreenDroid to analyze each of our application subjects multiple times and studied how the code coverage would change accordingly. Specifically, GreenDroid analyzed each application nine times. For these nine runs, we gradually increased the length limit from zero to eight and measured the percentage of source code lines that were executed (i.e., statement coverage). We chose statement coverage as the metric for measuring the thoroughness of our diagnosis for two reasons. First, to the best of our knowledge, we are not aware of any existing metrics that are designed for assessing the thoroughness of energy diagnosis. Second, statement coverage has been widely used for measuring code coverage for general purposes because it strikes a good balance between utility and collection overheads [40, 81]. Table 3.10 reports our study results and from them we obtain two major findings as discussed in the following.

Coverage saturation. We observe that for all application subjects, the statement coverage increases quickly at the beginning with the growth in the length of

Table 3.10: Statement coverage with respect to different event sequence length limits

Application name	# activities	Statement coverage (%) w.r.t different event sequence length limits (0 to 8)								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
DroidAR	6	0.54[1]	2.28	11.99	11.99	**12.54**	12.54	12.54	12.54[2]	12.54
Recycle Locator	3	1.23	16.11	23.76	28.17	32.18	**36.96**	36.96	36.96	36.96
Ushahidi	17	1.47	4.06	10.97	15.17	19.87	25.35	**25.39**	25.39	25.39
AndTweet	6	1.74	10.25	12.07	**15.94**	15.94	15.94	15.94	15.94	15.94
Ebookdroid	8	0.20	2.02	2.79	12.72	**25.81**	25.81	25.81	25.81	25.81
BableSink	1	2.68	24.39	30.33	**30.38**	30.38	30.38	30.38	30.38	30.38
CWAC-Wakeful	1	1.12	10.27	32.37	**42.30**	42.30	42.30	42.30	42.30	42.30
Sofia Public Transport Nav.	3	3.47	9.70	24.67	37.91	**38.12**	38.12	38.12	38.12	38.12
Osmdroid	8	1.01	11.36	18.09	18.93	24.68	**30.15**	30.15	30.15	30.15
Zmanim	3	1.72	11.81	27.71	27.96	28.04	**28.08**	28.08	28.08	28.08
Geohash Droid	9	2.96	10.31	19.87	22.94	25.58	**25.62**	25.62	25.62	25.62
My Tracks	12	1.06	5.30	12.72	20.17	**23.56**	23.56	23.56	23.56	23.56
Omnidroid	16	0.45	8.64	17.91	18.25	**20.88**	20.88	20.88	20.88	20.88
GPSLogger	1	4.86	14.11	44.31	**46.13**	46.13	46.13	46.13	46.13	46.13

[1]: Statement coverage is not 0 because in our implementation we do not count "launch the entry activity (when analysis starts)" and "finish all active activities and services (when analysis ends)" when generating user interaction event sequences.

[2]: Underlined runs took more than one hour to finish. Memory consumption (maximum heap size set to 4GB) did not increase much when we relaxed the length limits.

generated event sequences. The coverage gradually saturates at certain points and stops increasing when the length limit further grows. Take Osmdroid as an example. Its statement coverage increases from 1.01% to 24.68% when the length limit grows from zero to four. When the length limit reaches five, the statement coverage saturates at 30.15%, with no further increase even if the length limit grows to a larger value. Other applications are similar. To understand why, we inspected all these applications. We found that many of these applications contain only a small number of activity components (with GUI). As listed in the second column of Table 3.10, 8 of our 14 applications contain no more than six activity components. Although the applications Ushahidi and Omnidroid contain relatively larger number of activity components, we found that many of these activity components are actually designed only for displaying information. Besides, for user friendliness, developers have made their applications' GUIs intuitive. This means that users do not have to perform very long sequences of interactions from an application's entry GUI to reach other GUIs for using their designed major functionalities. This explains why the statement coverage measurement can quickly saturate for our studied applications.

Difficulties in achieving high coverage. We also observe that even if our event sequence generation enumerates all possible combinations of user interaction events, GreenDroid can still achieve only low statement coverage for some applications. For example, for DroidAR, AndTweet, My Tracks and Omnidroid, GreenDroid covers less than 25% statements. We thus inspected these applications and found three major difficulties in achieving higher code coverage. These findings can benefit related research such as automated Android application testing [40, 55]. We discuss these findings in the following:

- **Sophisticated external stimulus.** Achieving high code coverage may require sophisticated external stimulus for certain Android applications. For example, Omnidroid registers a broadcast receiver with Android OS to monitor 26 different system broadcast events (e.g., "missing phone call" and "phone connected to a physical dock" broadcast events). A large proportion of its code is used for handling such broadcasted system events, while our GreenDroid currently cannot actively generate such events. This suggests that in

order to cover such code, systematic simulation of external stimulus would be necessary.

- **Complex inputs and non-standard user interactions.** Achieving high code coverage may require complex inputs and non-standard user interactions for certain Android applications. Take DroidAR, an augmented reality application on Android, for example. It presents its user a live view of real-world objects that are augmented with various sensory inputs, and allows the user to interact with these objects digitally. In many cases, DroidAR requires video input from phone cameras for recognizing and rendering augmented objects accordingly. It contains two types of GUI elements: (1) standard GUI elements defined in Android libraries (e.g., buttons), and (2) augmented objects rendered by native graphics libraries. Both types of GUI elements can be dynamically updated. Therefore, covering a high proportion of DroidAR code would require its user not only to interact with standard GUI elements (e.g., clicking buttons), but also to interact with the non-standard GUI elements (e.g., rotating augmented objects). However, our GreenDroid currently cannot support video inputs or user interactions with non-standard GUI elements. This explains why GreenDroid achieves low code coverage when diagnosing DroidAR.

- **Special running environment.** Achieving high code coverage may require special running environments for certain Android applications. For example, AndTweet is a light-weight Twitter chat client. Covering most of its code requires: (1) a valid Twitter account, (2) network connectivity, and (3) meaningful data (e.g., tweets and followers) associated with this account. Failing to satisfy any of these requirements would make the application run meaninglessly, leading to low code coverage. Our GreenDroid currently does not know how to satisfy such application-specific requirements and this deserves further research.

From the above discussions, we can make two observations. First, similar to related studies [38], it is practical to limit the length of generated event sequences in

program analysis due to the combinatorial explosion problem. In our case, setting the length limit to six is a cost-effective choice. This is because a larger length limit does not further improve code coverage, but instead results in much longer diagnosis time (as in a magnitude of hours), as reported by our experiments. In practice, such settings should be made on a case by case basis as different applications may have different characteristics. Therefore, tools like our GreenDroid should allow its users to customize their required depth of diagnosis and provide a time budget [73]. Second, we observed that for some application subjects, GreenDroid located their energy problems even with low statement coverage. This can be explained. As discussed earlier (Section 3.1), energy problems typically only occur at certain application states reached by handling corresponding user interactions. For example, the energy problem in Zmanim can be exposed by the following four steps: (1) switching on GPS, (2) configuring Zmanim to use current location, (3) starting Zmanim's main activity, and (4) hitting the "Home" button when GPS is acquiring a location. Therefore, generating user interactions in a certain order is a prerequisite for exposing such problems. GreenDroid essentially enumerates all possible combinations of different types of user interaction events (e.g., button click events and checkbox selection events) and provides appropriate event values when generating these events. This explains why it can systematically explore an application's state space to locate potential energy problems. This also suggests that although statement coverage can be used for measuring the code coverage achieved by a certain energy diagnosis approach, it may not be a good metric for assessing the effectiveness of such energy diagnosis.

3.3.5 Comparison with Resource Leak Detection Work

Our work shares some similarity with existing resource leak detection work [39, 53], [92, 94] since sensor listeners and wake locks are considered as valuable resources in Android OS and applications. Our research question RQ7 studies how our GreenDroid compares to such work in terms of detecting real missing sensor or wake lock deactivation problems. To answer this question, we chose Relda for comparison [53]. Relda is the latest resource leak detection work dedicated for Android

applications. It is a fully automated static analysis tool for Android applications and supports detecting leak of 65 types of system resources, which also include sensor listeners and wake locks as studied in our work. Therefore, it would be interesting to know whether Relda can also effectively help detect missing sensor or wake lock deactivation problems in our studied Android application subjects. With the help of Relda's authors, we conducted experiments using their original tool (not our implementation, which can otherwise lead to bias in the comparison). In the experiments, we successfully applied Relda on 13 of our application subjects listed in Table 3.8 (except My Tracks). Relda reported 36 resource leak warnings, out of which 15 are related to sensors and wake locks, while the remaining 21 are related to other seven types of resources (e.g., phone cameras), which are outside the scope of our study. We further invited Relda's authors to manually validate these raw data and remove duplicate and false warnings as they did in their publication [53] (we did not do it by ourselves in order to avoid bias). Finally, they confirmed that Relda detected two real resource leak problems in DroidAR and one in Ebookdroid out of the 13 application subjects. By analyzing the experimental results, we obtained several findings as discussed below.

First, the two problems Relda detected in DroidAR happened because developers forgot to unregister a sensor listener and to disable a phone vibrator after usage, respectively. The other problem Relda detected in Ebookdroid happened because developers forgot to recycle a velocity tracker (it tracks the velocity of touch events for detecting gestures like flinging) back to the Android OS after using it. From these results, we can see that Relda can indeed detect more types of resource leaks than GreenDroid since it has a much wider focus. However, two of the three detected real problems are not related to sensors or wake locks. Within the scope of our study, Relda actually detected only one real problem of our interest (i.e., the missing sensor deactivation problem in DroidAR). As a comparison, our GreenDroid detected eight missing sensor or wake lock deactivation problems in these 13 application subjects as we discussed earlier. All these eight problems (including the one detected by Relda) are real problems as confirmed by developers.

Second, we carefully studied Relda to understand why it cannot effectively detect the other seven real missing sensor or wake lock deactivation problems that can be detected by GreenDroid in our studied Android applications. Based on our study results and our communications with Relda's authors, we identified four major reasons: (1) Relda does not conduct intra-procedural flow analysis. To avoid false positives, which can be a major concern for static analysis, Relda does not report any resource leak problem as long as a concerned resource can possibly be released at any program path. Due to this conservative nature, Relda did not effectively detect missing wake lock deactivation problems in BabbleSink and AndTweet. For example, the wake lock acquired by AndTweet might be released in certain program paths, but such paths could only be executed in exceptional cases that are not feasible during normal running (see Section 3.1.4 for more details). As such, AndTweet can constantly drain a phone's batter energy during its normal usage, but this problem cannot be reported by Relda. (2) Relda does not conduct points-to analysis. Thus it cannot figure out what object(s) a reference is pointing to, and this is a common limitation of static analysis techniques without points-to analysis. Due to this reason, Relda did not effectively detect the missing sensor deactivation problem in Ushahidi, where its developers mistakenly passed a newly created GPS sensor listener to the unregistration API (Line 11 in Figure 3.8) instead of passing the listener that has been registered earlier (Line 6 in Figure 3.8). (3) Relda does not properly model or consider event handler scheduling as we studied in this work. Thus it cannot handle message passing and receiving well. Due to this reason, it did not detect the missing wake lock deactivation problem in CWAC-Wakeful. The reason is that CWAC-Wakeful acquires a wake lock from the Android OS only when it receives a message that asks it to perform some long running task at background. (4) Relda did not detect missing sensor or wake lock deactivation problems in Recycle Locator, Sofia Public Transport Nav. and Ebookdroid due to its incomplete resource operation table. These applications use sensors or wake locks by calling compound APIs that wrap basic sensor listener registration/unregistration APIs or basic wake lock acquisition/releasing APIs. For example, Sofia Public Transport Nav. calls Google Maps APIs to use a phone's GPS sensor, and EbookDroid calls

the `setKeepScreenOn()` API in the `android.view.SurfaceView` class to acquire wake locks. Our GreenDroid does not have these discussed issues. It systematically executes an Android application. Its dynamic analysis is naturally flow-sensitive and does not need points-to analysis. Besides, it relies on our AEM model to ensure reasonable scheduling of event handlers so that it can handle messaging passing and receiving properly. Moreover, GreenDroid only focuses on two types of resources, i.e., sensor listeners and wake locks, so that we could prepare a more complete operation table for them with affordable effort. This explains why Relda missed some missing sensor or wake lock deactivation problems but GreenDroid could still detect them.

Third, although Relda can detect energy problems caused by missing sensor or wake lock deactivation as a form of resource leak, it cannot help diagnose energy problems caused by sensory data underutilization. These problems are more complicated as discussed throughout this chapter. Our GreenDroid supports automated analysis of sensory data utilization and can help developers diagnose energy problems caused by cost-ineffective use of sensory data.

From the above discussions, we can observe that both Relda and GreenDroid have their own scopes and strengths. Relda can detect a much wider range of resource leak problems and some of them may lead to serious energy waste. On the other hand, GreenDroid's scope is more focused (sensor and wake lock related energy problems) and its energy problem detection capability is satisfactory. In terms of detecting energy problems caused by missing sensor or wake lock deactivation, GreenDroid performs better than Relda. We did not compare GreenDroid to other resource leak detection work due to various reasons including tool availability and applicability (some work are for conventional Java programs, e.g., Torlak et al.'s work [92]). The above comparisons and discussions confirm that GreenDroid is useful and effective for diagnosing energy problems in Android applications, and its idea may also complement and contribute to existing resource leak detection work on the Android platform.

Table 3.11: Energy saving case study result

Experiment Time	Application version	Collected GPS points	Discarded GPS points	Discarding rate	Energy consumption (Joule)
Day 1 (1pm ~ 5pm)	GPSLogger-r15	266	127	32.3%	CPU: 709.7J; GPS: 4842.6J
Day 3 (1pm ~ 5pm)	GPSLogger-r15	304	135	30.8%	CPU: 835.2J; GPS: 5626.5J
Day 5 (1pm ~ 5pm)	GPSLogger-r15	272	144	34.6%	CPU: 761.4J; GPS: 5019.3J
Day 2 (1pm ~ 5pm)	GPSLogger-clean	230	51	18.1%	CPU: 601.4J; GPS: 4217.1J
Day 4 (1pm ~ 5pm)	GPSLogger-clean	253	63	19.9%	CPU: 625.3J; GPS: 4354.6J
Day 6 (1pm ~ 5pm)	GPSLogger-clean	293	58	16.5%	CPU: 680.5J; GPS: 4660.7J

3.3.6 Energy Saving: A Case Study

To answer research question RQ8 about potential energy saving if our detected energy problems can be fixed, we conducted a real case study. In practice, users may interact with an application in different ways, and this could affect the application's energy consumption quite significantly [44]. To minimize the effect of such user interactions, we selected the application GPSLogger as our case study subject because it requires almost no human intervention after initial setup. We prepared two versions of GPSLogger: one with energy problem (will be referred to as GPSLogger-r15) and the other with our patch (will be referred to as GPSLogger-clean). We built this patch conservatively by slightly modifying the GPS sensing part of GPSLogger. To be realistic, we built this patch by following Geohash Droid's real patch on fixing its energy problem (issue 24) [16]. Specifically, the patched GPSLogger would slow down its GPS sensing rate to every 30 seconds when it finds its collected GPS data keep being of low quality (e.g., after five consecutive imprecise readings), and set the sensing rate immediately back to the original value when it finds that GPS data have become precise again (e.g., after two consecutive precise readings). In fact, one can also do it in an aggressive way by disabling GPS sensing or setting a longer slow-sensing period if the application keeps receiving imprecise GPS data. Although this may save more energy, we took the previous conservative strategy for making our best efforts in avoiding potential effect on the application's functionality.

Table 3.11 compares the energy consumption between the two versions of GPSLogger. We conducted the case study for six consecutive days. On each day, our study participant, a postgraduate student, who was unaware of our experimental setup and purpose, strictly followed the same pre-specified activity pattern: (1) walking from his office to canteen and having lunch there (from 1:00 pm to 1:45 pm), (2) walking back to his office and then studying (from 1:45 pm to 3:30 pm), (3) walking to library and read some newspapers or magazines (from 3:30 pm to 4:15 pm), and (4) finally going to gym for physical exercises (from 4:15 pm to 5:00 pm). This activity pattern is common for a postgraduate student. We had this participant carry the same smartphone, Samsung Galaxy S3 (with Android 4.1.2), with different versions of GPSLogger installed on different days (he is unaware of this difference), as shown in Table 3.11. For each version, we arranged three days for experiments, to minimize effects of unpredictable and uncontrollable physical environment (e.g., GPS signal strength may be subject to change for unknown reasons). At the end of each day, we collected the following information from the smartphone: (1) number of precise GPS points collected, (2) number of imprecise GPS points discarded, and (3) energy consumed by GPSLogger during the experiment. We measured energy consumption by PowerTutor [102], which is a highly rated tool for measuring real energy consumption (in Joules) for selected Android applications or components.

Table 3.11 reports our study results. We give energy consumption data only for CPU and GPS sensor in each day's experiment. This is because GPSLogger ran at background and thus battery energy was mostly consumed by its CPU computing and GPS sensing. We can make two observations from Table 3.11. First, in six experiments, the two versions of GPSLogger collected comparable numbers of GPS data points, ranging from 230 to 304 with a mean of 270. Since GPSLogger is mainly designed for recording its user's location traces, such small difference has little effect on the application's functionality. In fact, our participant also did not notice any difference in terms of user experience while using the two versions of this application. Second, we observe a large drop in GPS data discarding rate for the patched version GPSLogger-clean. On average, GPSLogger-clean discarded 18.2% of GPS data, while GPSLogger-r15 discarded 32.6% of GPS data (79.1% more).

Accordingly, GPSLogger-clean consumed 635.7J in CPU computing and 4410.8J in GPS sensing. For GPSLogger-r15, the energy consumption became 768.8J in CPU computing (20.9% more) and 5162.8J in GPS sensing (17.0% more). Note that this comparison is based on a conservative strategy, and in practice, the difference can be even larger (e.g., if the patch adopts an aggressive strategy). This shows that fixing GreenDroid's detected energy problem can indeed save much energy consumption on real smartphones.

With these promising results, we submitted our patch to GPSLogger developers. The patch was recently accepted. We also helped release it online for trial downloads for interested users.[18] So far, this patch has received around 1,000 downloads. This indicates that developers indeed acknowledge and accept our efforts in helping defend their Android applications from energy inefficiency.

3.3.7 Discussions

Tool implementation. Our energy diagnosis approach is independent of its underlying program analysis framework. Currently, we implemented it on top of JPF because JPF is a highly extensible Java program verification framework with internal support for dynamic tainting. However, analyzing Android applications using JPF is still challenging as discussed throughout this chapter. We have to carefully address the problems of event sequence generation and event handler scheduling, as well as Android library modeling. In particular, modeling Android libraries is known to be a tedious and error-prone task [73]. This is why our current implementation only modeled a partial, but critical, set of library classes and concerned APIs. Extending our tool to support more Android APIs is possible, but would require more engineering effort, and our GreenDroid is evolving along this direction. Besides, in GreenDroid's current implementation, all temporal rules in our AEM model have been translated into code for ease of use. We are considering building a more general execution engine that can take these rules as inputs to schedule Android event handlers reasonably. This would make our GreenDroid more extensible to new rules.

[18] `https://code.google.com/p/gpslogger/downloads/list`

To realize this, we need: (1) a new domain language to specify these rules, and (2) a mechanism that automatically interprets and enforces these rules at runtime. Moreover, we are also considering integrating our diagnosis approach into Android framework by modifying the Dalvik virtual machine much the same as Enck et al. did [48]. This can bring two benefits. First, it enables real-time energy inefficiency diagnosis. Second, modeling Android libraries is no longer necessary, such that the imprecision caused by inadequate library modeling can also be alleviated or avoided. Lastly, GreenDroid can be designed to be interactive, providing its users visualizations of sensory data usage details. This would help developers quickly figure out the root causes for a wide range of domain-specific energy problems.

Tainting quality. Our sensory data utilization analysis relies on dynamic tainting for tracking propagation of sensory data. It is well known that designing precise dynamic tainting is challenging [90]. Researchers have found that ignoring control dependencies in taint propagation can cause *undertainting* (i.e., failing to taint some data derived from taint sources), but considering control dependencies can also cause *overtainting* (i.e., tainting some data that are not derived from taint sources) [61]. It is therefore suggested that the tainting policy should be designed according to its application scenarios [90]. In our case, we need to track propagation of sensory data and identify program data that are derived from such sensory data. For this purpose, we adapted TaintDroid's tainting policy [48] and added a special rule for handling control dependencies (ignoring control dependencies is one of TaintDroid's limitations). While this rule may potentially result in overtainting in theory, we did not observe any evident impact on our sensory data utilization analysis results. We made some analysis of our studied application subjects. We found that unlike user privacy data (e.g., phone number) handled by TaintDroid, sensory data in our studied applications are typically updated frequently. These data can be quickly replaced with new data. Their consumption is thus short-term, implying that they are unlikely to affect a large volume of program data in Android applications. This explains why our control dependency handling does not introduce evident overtainting problems.

Limitations. Our current GreenDroid implementation has some limitations. First, GreenDroid cannot generate complex inputs (e.g., video inputs or user gestures). Thus, there can be application states not reachable by GreenDroid. If any energy problem is associated with these states, GreenDroid would not be able to detect them (i.e., the analysis may be incomplete, leading to false negatives in bug detection). Second, GreenDroid's event sequence generation belongs to the category of model-based approaches [55, 73, 98]. One common problem with these approaches is that they rely on a statically extracted model and lack runtime information. For example, GreenDroid relies on a GUI model extracted by statically analyzing an application's layout configurations. It cannot cope with dynamic GUI updates (e.g., news reading applications can dynamically load a new list of items). Therefore, we found in our evaluation that GreenDroid sometimes generated infeasible user interaction event sequences (e.g., a sequence containing a click event on a GUI element that has been removed). For our largest subject DroidAR, GreenDroid generated around 8% infeasible event sequences due to its inability to handle dynamic GUI updates. Because of this limitation in event sequence generation, our analysis may be unsound in certain cases, leading to false positives in bug detection. Third, GreenDroid cannot systematically simulate different sensory data as this requires a comprehensive characteristic study of real-world sensory data. Currently, we randomly picked up mock sensory data from a pre-prepared data pool controlled by different precision levels. It could be possible that the selection of sensory data has an impact on a program's control flow (e.g., an execution path that requires specific data values cannot be explored). Although we did not observe the above three issues affecting GreenDroid's effectiveness in diagnosing our application subjects, we are investigating them and plan to come up with more complete solutions in future. For example, the second limitation may be addressed by integrating GreenDroid's energy inefficiency diagnosis into the Android framework. Then its event sequence generation no longer needs pre-extracted GUI models for Android applications under diagnosis. Instead, one can analyze an application's GUI layout at runtime and adapt automated testing tools like Robotium [30] for generating user interaction events. This limitation may also be addressed by adding event sequence feasibility

validation to GreenDroid (e.g., using Jensen et al.'s work [55]). Then GreenDroid can first validate the feasibility of its generated event sequences before presenting them to developers for reproducing its detected energy problems. We leave these potential improvements to our future work.

Alternative analysis approach. Our current sensory data utilization analysis is only one possible approach. It analyzes how many times and to what extent sensory data are utilized by an application at certain states. We believe that there can also be other good designs for effective analysis of sensory data utilization. We discuss one possible alternative here. For example, instead of accumulating sensory data consumptions (i.e., analyzing how many times sensory data are utilized; see Equation 3.2) in the analysis, we can also consider that as long as sensory data are effectively utilized once, the battery energy for collecting the data is well spent. Besides, when designing the "data usage" metric, we can also choose not to distinguish different APIs that utilize sensory data. Specifically, we can choose not to scale the usage metric value by the number of bytecode instructions executed during the invocation of an API that utilizes sensory data (i.e., not analyzing to what extent the sensory data are utilized). Such a design may also help locate energy problems. For instance, although we cannot distinguish how many times sensory data are utilized in different application states, we can still identify application states that totally do not utilize sensory data. In our experiments, we found that such "complete energy waste" cases indeed exist (i.e., GPSLogger's energy problem). However, for most of our studied energy problems, the concerned applications do not totally discard collected sensory data. For example, Geohash Droid always uses location data to update a phone's notification bar (see Figure 3.3(a)), but still its developers consider that if other remote listeners are not actively monitoring location updates, then only updating phone notification bar is a waste of valuable battery energy. In such cases, the alternative design might not be able to locate such energy problems. As a comparison, our approach can not only help locate application states that totally do not utilize sensory data, but also help locate those that do not utilize sensory data in a fully effective manner. Therefore, it can generally provide finer-grained information for energy diagnosis and optimization. Of course,

our design allows GreenDroid to report more energy problems than the alternative design. This is why we also propose a prioritization strategy to help developers focus on the potentially most serious energy problems, i.e., those have the lowest data utilization coefficients.

Fixing detected energy bugs. Our GreenDroid can help detect three common patterns of energy bugs. To fix the detected missing sensor or wake lock deactivation bugs, developers can simply add sensor listener unregistration and wake lock releasing operations in the corresponding program locations. However, fixing sensory data underutilization bugs is a non-trivial task for developers. In our empirical study, we observed two commonly-used strategies: (1) temporarily reducing sensing frequency when sensory data are not utilized in a fully-effective manner, and (2) temporarily deactivating sensors when sensory data are completely not useful. Based on this observation, GreenDroid would suggest developers to consider the two optimization strategies when it detects energy bugs caused by sensory data underutilization. Nevertheless, how to fix detected sensory data underutilization bugs should be application-specific. In particular, if developers choose to follow the second strategy, which is more aggressive than the first one, they should take into account that activating and deactivating sensors have energy overhead. In other words, developers need to carefully judge whether the energy consumed by the useless sensing operations in the inefficient version of their application outweighs the energy cost of deactivation (when the sensory data are completely not useful) and reactivation of sensors (when the application reaches a state where sensory data will be effectively utilized). If yes, the optimization can bring overall energy saving. Otherwise, they should consider to optimize their application using other strategies.

3.4 Related Work

Our GreenDroid work relates to several research topics, which include energy efficiency analysis, energy consumption estimation, resource leak detection, and information flow tracking. Some of them particularly focus on smartphone applications. In this section, we discuss representative pieces of work in recent years.

3.4.1 Energy Efficiency Analysis

Smartphone applications' energy efficiency is vital. In past several years, researchers have worked on this topic mostly from two perspectives. First, various design strategies have been proposed to reduce energy consumption for smartphone applications. For example, MAUI [43] helped offload "energy-consuming" tasks to resource-rich infrastructures such as remote servers. EnTracked [62] and RAPS [80] adopted different heuristics to guide an application to use GPS sensors in a smart way. Little Rock [85] suggested a dedicated low power processor for energy-consuming sensing operations. SALSA [88] helped select optimal data links for saving energy in large data transmissions. Second, different techniques have been proposed to diagnose energy problems in smartphone applications. Kim et al. proposed to use power signatures based on system hardware states to detect energy-greedy malware [60]. Pathak et al. conducted the first study of energy bugs in smartphone applications, and proposed to use reaching-definition dataflow analysis algorithms to detect no-sleep energy bugs, which can arise from mishandling of power control APIs in Android applications (e.g., wake lock acquisition/releasing APIs) [82, 84]. Zhang et al. proposed a taint-tracking technique for the Android platform to detect energy wastes caused by unnecessary network communications [101]. To help end users troubleshoot energy problems on their smartphones, Ma et al. built a tool to monitor smartphones' resource usage behavior as well as system or user events (e.g., configuration changes in certain applications) [70]. Their tool can help identify triggering events that cause abnormally high energy consumption, and suggest corresponding repair solutions (e.g., reverting configuration changes) to users.

Our work shares a similar goal with these pieces of work, in particular, recent work in the second category discussed above [70, 84, 101]. Nevertheless, our work differs from them on several aspects. Regarding Pathak et al.'s work [101], our work has two distinct differences. First, we found that detecting no-sleep bugs like missing wake lock deactivation is not difficult. One can always adapt existing resource leak detection (as we did in our work) or classic reaching-definition data flow analysis (as they did in their work) techniques for this purpose. However, our empirical

study revealed more subtle energy problems caused by sensory data underutilization. As discussed earlier, effectively detecting sensory data underutilization problems is non-trivial. It requires a systematic exploration of an application's state space and a precise analysis of sensory data utilization. Second, to conduct data flow analysis, Pathak et al. assumed that control flows between event handlers were already available from application developers. This is not a practical assumption for Android applications. Asking developers to manually derive program control flow information is unrealistic, especially when applications contain hundreds of event handlers (e.g., our experimental subjects DroidAR and Omnidroid). As such, we chose to formulate handler scheduling policies extracted from Android specifications as an AEM model so that it can be reusable across different applications for correctly scheduling event handlers during program analysis. Our experimental results have confirmed that this model is necessary and useful for effectively diagnosing energy problems in Android applications.

Zhang et al.'s work also makes a similar observation to ours, i.e., using network data to update invisible GUIs can be an energy waste [101]. However, our work differs from theirs in three ways. First, we focus on energy problems caused by cost-ineffective uses of sensory data instead of network data, as our empirical study reveals that ineffective use of sensory data has often caused massive energy waste. Second, besides analyzing how sensory data are utilized by Android applications, we also studied ways of systematically generating event sequences to exercise an application, while their work may require extra testing effort for effective analysis (they did not study how to automate an application's execution for analysis). Third, we proposed a state-based analysis of sensory data utilization. It effectively distinguishes different usage scenarios of sensory data, while Zhang et al.'s work only supports distinguishing two types of scenarios, i.e., network data used to update visible or invisible GUIs, respectively. As a result, our work can provide richer information to help diagnose energy problems with a wider scope.

Our work also has a different objective from Ma et al.'s work [70]. Their work does not analyze an application's program code. Instead, it monitors a device's energy consumption as well as system or user events to help identify those events

that have likely caused abnormally high energy consumption. By reverting the effect of these events (e.g., uninstalling a suspicious application), users can potentially suffer less battery drain. On the other hand, our work directly diagnoses causes of energy problems in an application's program code and helps fix them by providing concrete problem-triggering conditions.

3.4.2 Energy Consumption Estimation

One major reason why so many smartphone applications are not energy efficient is that developers lack viable tools to estimate energy consumption for their applications. Extensive research has been conducted to address this topic. PowerTutor [102] uses system-level power-consumption models to estimate the energy consumed by major system components (e.g., display) during the execution of Android applications. Such models are a function of selected system features (e.g., CPU utilization) and obtained by direct measurements during the controlling of the device's power state. Sesame [46] shares the same goal as PowerTutor, but can perform energy estimation for much smaller time intervals (e.g., as small as 10ms). eProf [83] is another estimation tool. Instead of estimating energy consumption at a system level like PowerTutor and Sesame, eProf estimates energy consumption at an application level by tracing system calls made by applications when they run on smartphones. WattsOn [74] further extends eProf's idea by enabling developers to estimate their applications' energy consumption on their workstations, rather than real smartphones. The most recent work is eLens [54]. It combines program analysis and per-instruction energy modeling to enable much finer-grained energy consumption estimation. However, eLens assumes that smartphone manufacturers should provide platform-dependent energy models for each instruction. This is not a common practice as both the hardware and software of a smartphone platform can evolve quickly. Requiring manufacturers to provide a new set of instruction-level energy models for each platform update is impractical. Regarding this, eLens provides a hardware-based technical solution to help obtain such energy models. Still, power measurement hardware may not generally be accessible for real-world developers.

Typical scenarios for the techniques discussed above are to identify hotspots (software components that consume the most energy) in smartphone applications, such that developers can perform energy consumption optimization. However, simply knowing the energy cost of a certain software component is not adequate for an effective optimization task. The missing key information is whether this energy consumption is necessary or not. Consider an application component that continually uses collected GPS data to render a map for navigation. This component can consume a lot of energy and thus be identified as a hotspot. However, although the energy cost can be high, this component is evitable in that it produces great benefits for its users by smart navigation. As such, developers may not have to optimize it. Based on this observation, our GreenDroid work helps diagnose whether certain energy consumed by sensing operations can produce corresponding benefits (i.e., high sensory data utilization). This can help developers make wise decisions when they face the choice of whether or not to optimize energy consumption for certain application components. For example, if they find that at some states, sensing operations are performed frequently, but thus collected sensory data are not effectively utilized, then they can consider optimizing such sensing mechanisms to save energy as Geohash Droid developers did [16] (issue 24).

3.4.3 Resource Leak Detection

System resources are finite and usually valuable. Developers are required to release acquired resources in a timely fashion for their applications when these resources are no longer needed. However, tasks for realizing this requirement are often error-prone due to a variety of human mistakes. Empirical evidence shows that resource leaks commonly occur in practice [94]. To prevent resource leaks, researchers proposed language-level mechanisms and automated management techniques [45]. Various tools were also developed to detect resource leaks [39, 92]. For example, QVM [39] is a specialized runtime environment for detecting defects in Java programs. It monitors application executions and checks for violations of resource safety policies. TRACKER [92] is an industrial-strength tool for finding resource leaks in Java programs. It conducts inter-procedural static analysis to ensure no resource

safety policy is violated on any execution path. Besides, Guo et al. recently collected a nearly complete table of system resources in the Android framework that require explicit release operations after usage [53]. Similar to our work, they also adapted the general idea of resource safety policy checking discussed in QVM [39] and TRACKER [92] for problem detection. The major differences between our work and these pieces of work are two-fold. First, we proposed to systematically explore an Android application's state space for energy problem detection. This requires addressing technical challenges in generating user interaction event sequences and scheduling event handlers. Second, we also focused on studying more complex energy problems, i.e., sensory data underutilization. As discussed throughout this chapter, detecting these energy problems requires precise tracking of sensory data propagation and careful analysis of sensory data usage. Regarding this, we have proposed analysis algorithms and automated problem detection in this work, and they have not been covered by these pieces of existing work.

3.4.4 Information Flow Tracking

Dynamic information flow tracking (DFT for short) observes interesting data as they propagate in a program execution [59]. DFT has many useful applications. For example, TaintCheck [75] uses DFT to protect commodity software from memory corruption attacks such as buffer overflows. It taints input data from untrustworthy sources and ensures that they are never used in a dangerous way. TaintDroid [48] prevents Android applications from leaking users' private data. It tracks such data from privacy-sensitive sources, and warns users when these data leave the system. LEAKPOINT [42] leverages DFT to pinpoint memory leaks in C and C++ programs. It taints dynamically allocated memory blocks and monitors them in case their release might be forgotten. Our GreenDroid work demonstrates another application of DFT. We showed that DFT can help track propagation of sensory data, such that their utilization analysis against energy consumption can be conducted to detect potential energy problems in smartphone applications.

3.5 Chapter Summary

In this chapter, we presented an empirical study of real energy problems in 402 Android applications, and identified two types of coding phenomena that commonly cause energy waste: missing sensor or wake lock deactivation, and sensory data underutilization. Based on these findings, we proposed an approach for automated energy problem diagnosis in Android applications. Our approach systematically explores an application's state space, automatically analyzes its sensory data utilization, and monitors the usage of sensors and wake locks. It helps developers locate energy problems in their applications and generates actionable reports, which can greatly ease the task of reproducing energy problems as well as fixing them for energy optimization. We implemented our approach into a tool GreenDroid on top of JPF, and evaluated it using 14 real-world popular Android applications. Our experimental results confirmed the effectiveness and practical usefulness of GreenDroid.

Chapter4

Characterizing and Detecting Performance Bugs

In this chapter, we focus on performance bugs in Android applications. We first present an in-depth empirical study of real performance bugs from popular Android applications in Section 4.1. The empirical study identified three common patterns of performance bugs. With the findings, we then present a light-weight static analysis technique to detect these performance bugs in Section 4.2. Next, we evaluate our technique on a large set of real-world Android applications in Section 4.3. After that, we discuss existing studies related to performance bug diagnosis in mobile applications in Section 4.4. Finally, we give a chapter summary in Section 4.5. The content of this chapter is based on two research papers [66, 67].

4.1 Characterizing Performance Bugs

In this section, we present our empirical study of real-world performance bugs from Android applications. The study aims to answer the following four research questions:

- **RQ1 (Bug types and impacts):** *What are common types of performance bugs in Android applications? What impacts do they have on user experience?*
- **RQ2 (Bug manifestation):** *How do performance bugs manifest themselves? Does their manifestation need special inputs?*
- **RQ3 (Debugging and bug-fixing effort):** *Are performance bugs more difficult to debug and fix than non-performance bugs? What information or tools can help with this?*

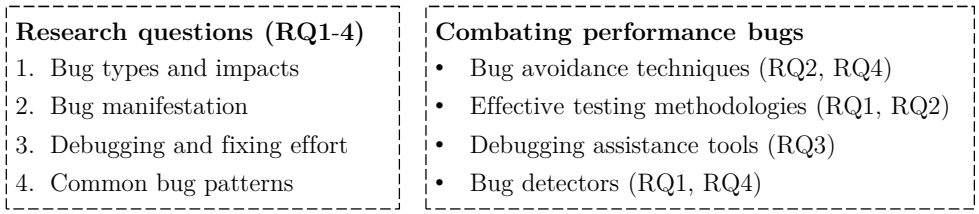

Figure 4.1: Potential benefits of our empirical findings

- **RQ4 (Common bug patterns):** *Are there common causes of performance bugs? Can we distill common bug patterns to facilitate automated performance analysis and bug detection?*

We formulated these research questions by studying related work [57, 99]. Through answering them, we not only aim to understand the characteristics of performance bugs in Android applications, but also wish to support follow-up research on performance bug avoidance, testing, debugging and detection for Android applications, as illustrated in Figure 4.1. For instance, with identified common bug patterns, one can propose guidelines for avoiding certain performance bugs in application development. One can also design and implement bug detection tools for identifying performance optimization opportunities in application testing and maintenance.

4.1.1 Study Methodology

We selected open-source Android applications as our subjects for studying research questions RQ1–4 because the study requires application bug reports and corresponding code revisions. 29 candidate applications that satisfy the following three criteria were randomly selected from four popular open-source software hosting platforms, namely, Google Code [18], GitHub [17], SourceForge [34], and Mozilla repositories [24].

- **Popularity.** First, a candidate application should have achieved more than 10,000 downloads on market (i.e., popular).

- **Traceability.** Second, a candidate application should own a public bug tracking system (i.e., traceable).

- **Maintainability.** Third, a candidate application should have at least hundreds of code revisions (i.e., well-maintained).

The above three criteria provide a good indicator of popular and mature applications. For these 29 candidates, we tried to identify performance bugs in their bug tracking systems. Due to different management practices, some application developers explicitly labeled performance bugs using special tags (e.g., "perf"), while others did not maintain a clear categorization of reported bugs. To ensure that we study real performance bugs, we refined our application selection by keeping only those containing clearly labeled performance bugs for our study. As a result, eight applications were finally selected as our subjects (all 29 applications were still used in our later evaluation for validating the usefulness of our empirical findings, see Section 4.3). From them, we obtained a total of 70 performance bugs, which were clearly labeled (confirmed) and later fixed.

Our selection process could miss some performance bugs (e.g., those not performance-labeled). Some related studies selected performance bugs by searching keywords like "slow" or "latency" in bug reports [57, 99]. We found that such searches resulted in more than 2,800 candidate performance bugs in all 29 applications. We randomly sampled and manually analyzed 140 of these candidate bugs (5%), and found that most of them are inappropriate for our study. This is because more than 70% of these candidates are either not related to performance (i.e., their bug reports accidentally contain such keywords) or are actually complex bugs that contain both performance and functional issues (e.g., low performance as a side effect of wrongly implemented functionality). To avoid introducing such threats or uncontrollable issues to our empirical study, we refrained from keyword search, while focusing on the 70 explicitly labeled performance bugs.

Table 4.1 lists the basic information of our eight selected Android applications. They are large-scale (up to 122.9K LOC), popularly-downloaded (up to 100 million downloads), and cover five different application categories. In the following, we analyze the 70 performance bugs collected from these applications and report our findings. The whole empirical study took about 180 person-days, involving three

Table 4.1: Subjects for studying performance bug characteristics

Application name	Category	Size (LOC)	Programming language	Downloads	Availability	# selected bugs
Firefox[2]	Communication	122.9K[1]	Java, C++, C	10M[1] ~ 50M	Mozilla Repositories	34
Chrome[2]	Communication	77.3K	Java, C++, Python	50M ~ 100M	Google Code	19
AnkiDroid	Education	44.8K	Java	500K ~ 1M	Google Code	4
K-9 Mail	Communication	76.2K	Java	1M ~ 5M	Google Code	3
My Tracks	Health & Fitness	27.1K	Java	10M ~ 50M	Google Code	3
c:geo	Entertainment	44.7K	Java	1M ~ 5M	GitHub	3
Open GPS Tracker	Travel & Local	18.1K	Java	100K ~ 500K	Google Code	2
Zmanim	Books & Reference	5.0K	Java	10K ~ 50K	Google Code	2

[1]: 1K = 1,000 & 1M = 1,000,000; [2]: For Firefox and Chrome, we counted only their lines of code specific to Android.

students (two postgraduate and one final-year undergraduate) for data collection, analysis and cross-checking.

4.1.2 Bug Types and Impact

We studied the bug reports and related discussions (e.g., comments and patch reviews) of the 70 performance bugs, and assigned them to different categories according to their major consequences. If a bug has multiple major consequences, we assigned it to multiple categories (so accumulated percentages can exceed 100%). We observed three common types of performance bugs in Android applications:

GUI lagging. Most performance bugs (53 / 70 = 75.7%) are GUI lagging bugs. They can significantly reduce the responsiveness or smoothness of the concerned applications' GUIs and prevent user events from being handled in a timely way. For example, in Firefox browser, tab switching could take up to ten seconds in certain scenarios (Firefox bug 719493 [15]). This may trigger the infamous "Application Not Responding (ANR)" error and cause an application to be no longer runnable, because Android OS would force its user to close the application in such circumstances (see Section 2.2 for more details).

Energy leak. The second common type of performance bugs (10 / 70 = 14.3%) is energy leak.[1] With such bugs, the concerned applications could quickly consume excessive battery power with certain tasks, which actually bring almost no benefits to users. For example, the energy leak in Zmanim (bug 50 [37]) made the application render invisible GUI widgets in certain scenarios, and this useless computation simply wasted valuable battery power. If an Android application contains serious energy leaks, its user's smartphone battery could be drained in just a few hours. For instance, My Tracks has received such complaints (bug 520 [25]):

> *"I just installed My Tracks on my Galaxy Note 2 and it is a massive battery drain. My battery lost 10% in standby just 20 minutes after a full charge."*

> *"This app is destroying my battery. I will have to uninstall it if there isn't a fix soon."*

Energy leaks in smartphone applications can cause great inconvenience to users. Users definitely do not want their smartphones to power off due to low battery, especially when they need to make important phone calls. As shown in the above comments, if an application drains battery quickly, users may switch to other applications that offer similar functionalities but are more energy-efficient. Such a "switch" can be common since nowadays users have many choices in selecting smartphone applications.

Memory bloat. The third common type of performance bugs (8 / 70 = 11.4%) is memory bloat, which can incur unnecessarily high memory consumption (e.g., Firefox bug 723077 [15] and Chrome bug 245782 [7]). Such bugs can cause "Out of Memory (OOM)" errors and application crashes. Even if a concerned application does not crash immediately (i.e., mild memory bloat), its performance can become unstable as Dalvik garbage collection would be frequently invoked, leading to degraded application performance.

---

[1]Interestingly, mobile developers consider energy bugs as a type of performance bugs, while in conventional softwares, performance bugs typically refer to those bugs that can significantly slow down a software or cause serious waste of computational resources (e.g., memory) [57].

These three performance bug types have occupied a majority of our studied 70 performance bugs (94.7%; some bugs belong to more than one type as aforementioned). There are also other types of bugs (e.g., those causing high disk consumption or low network throughput), but we observed them only once for each type in our dataset. Thus, we consider them not common.

Based on these findings, we answer our research question RQ1: *GUI lagging, energy leak and memory bloat are three dominant performance bug types in our studied Android applications. Research effort can first be devoted into designing effective techniques to combat them.*

4.1.3 Bug Manifestation

Understanding how performance bugs manifest in Android applications can provide useful implications on how to effectively test performance bugs. Our study reveals some observations, which demonstrate unique challenges in such performance testing.

Small-scale inputs suffice to manifest performance bugs. Existing studies reported that two thirds of performance bugs in PC applications need large-scale inputs to manifest [57]. However, in our study, we observed only 11 performance bugs (out of 70) that require large-scale inputs to manifest. Here, we consider a database with 100 data entries already large-scale (e.g., Firefox bug 725914 [15]). Other bugs can easily manifest with small-scale inputs. For example, Firefox bugs 719493 and 747419 [15] only need one user to open several browser tabs to manifest. Manifested bugs would significantly slow down Firefox and make its GUI less responsive. We give some comments from their bug reports below:

> *"I installed the nightly version and found tab switching is so slow that it makes using more than one tab very hard."*

> *"Firefox should correctly use view holder patterns. Otherwise, it will just have pretty bad scrolling performance when you have more than a couple of tabs."*

These comments suggest that Android applications can be susceptible to performance bugs. If an application has issues affecting its performance, users can often have uncomfortable experiences when conducting simple daily operations like adding a browser tab (Firefox bug 719493 [15]). A few such operations can quickly cause performance degradation. Due to this reason, cautious developers should try their best to optimize the performance of their code. For example, c:geo developers always try to avoid creating short-term objects (c:geo bug 222 [6]), because Android documentation states that less object creation (even an array of Integers) means less garbage collection [67].

Special user interactions needed to manifest performance bugs. More than one third (25 out of 70) of performance bugs require special user interactions to manifest. For example, Zmanim's energy leak needs the following four steps to manifest: (1) switching on GPS, (2) configuring Zmanim to use current location, (3) starting its main activity, and (4) hitting the "Home" button when GPS is acquiring a location. Such bugs are common, but can easily escape traditional testing. They can only manifest after a certain sequence of user interactions happen to the concerned application, but traditional code-based testing adequacy criteria (e.g., statement or branch coverage) do not really consider sequences of user interactions. A recent study also shows that existing testing techniques often fail to reach certain parts of Android application code [55]. Hence, our findings suggest two challenges and corresponding research directions in testing performance bugs for smartphone applications:

- Effectively testing performance bugs requires coverage criteria that explicitly consider sequences of user interactions in assessing the testing adequacy. Since the validity of user interaction sequences is essentially defined by an application's GUI structure, existing research on GUI testing coverage criteria [72] may help in addressing this challenge.

- Test input generation should construct effective user interaction sequences to systematically explore an application's state space. Since such sequences can be infinite, research effort should focus on effective techniques that can identify

equivalence among constructed user interaction sequences, avoiding redundant sequences and wasted test efforts.

Automated performance oracle needed. Performance bugs can gradually degrade an application's performance. For example, Firefox becomes progressively slower when its database's size grows (bug 785945 [15]). Such bugs rarely cause fail-stop consequences like application crashes, thus it is challenging to decide whether an application is suffering from any performance bug. Yet, our study found three common judgment criteria that have been used in real world to detect performance bugs in Android applications:

- **Human oracle.** More than half of the judgments were made manually by developers or users in our investigated Android applications. People simply made judgments according to their own experiences.

- **Product comparison.** Many developers compared different products of similar functionalities to judge whether a particular product contains any performance bugs (e.g., checking whether conducting an operation in one product is remarkably slower than in other products). We observed ten such cases in our study. For example, upon receiving user complaints about performance, K9 Mail developers checked whether their application's performance was comparable to other email clients and then decided what to do next (K9 Mail bugs 14 and 23 [23]).

- **Developers' consensus.** Developers also have some implicit consensus for judging performance bugs. For instance, Google developers consider an application sluggish (i.e., GUI lagging) if a user event cannot be handled within 200 milliseconds [97]. Mozilla developers assume that Firefox's graphics-rendering units should be able to produce 60 frames per second to make smooth animations (Firefox bugs 767980 and 670930 [15]).

Although these judgment criteria have been used in practice, they either require non-trivial manual effort (thus not scalable) or are not generally defined (thus not widely used). To facilitate performance testing and analysis, automated oracles are

desirable. Even if general oracles may not be possible, application or bug specific oracles can still be helpful. Encouragingly, there have been initial attempts toward this end [84, 101]. Besides, in this thesis, we also proposed a cost-benefit analysis to detect energy leaks caused by improper or ineffective uses of smartphone sensors (see Section 3.2.4). Still, more effort on general automated oracles for performance bugs is needed to further advance related research.

Performance bugs can be platform-dependent. We also observed that a non-negligible proportion (6 out of 70) of performance bugs require specific software or hardware platforms to manifest. For example, Chrome's caching scheme would hurt performance on ARM-based devices, but not on x86-based devices (Chrome bugs 170344 and 245782 [7]). Firefox's animation works more smoothly on Android 4.0 than older systems (Firefox bug 767980 [15]). This suggests that developers should consider device variety during performance testing, since Android OS can run on different hardware platforms and has so many customized variants. This feature differs largely from performance bugs in PC applications, which are not so platform-dependent [57, 99].

Based on these findings, we answer our research question RQ2: *Effective performance testing needs: (1) new coverage criteria to assess testing adequacy, (2) effective techniques for generating user interaction sequences to manifest performance bugs, and (3) automated oracles to judge performance degradation.*

4.1.4 Debugging and Bug-Fixing Effort

To understand the effort required for performance debugging and bug-fixing for Android applications, we analyzed 60 of our 70 performance bugs. We excluded 10 remaining bugs because we failed to recover links between their bug reports and code revisions.[2] To quantify debugging and bug-fixing effort for each of these bugs, we measured three metrics that were also adopted in related studies [99]: (1) *bug open duration*, which is the amount of time from a bug report is opened to

[2]Our manual analysis of commit logs around bug-fixing dates also failed to find corresponding code revisions.

Table 4.2: Performance bug debugging and fixing effort

Metric	Min.	Median	Max.	Mean
Bug open duration (days)	1	47	378	59.2
Number of bug comments	1	14	71	16.7
Patch size (LOC)	2	72	2,104	182.3

the concerned bug is fixed; (2) *number of bug comments*, which counts discussions among developers and users for a bug during its debugging and bug-fixing period; (3) *patch size*, which is the lines of code changed for fixing a bug. Intuitively, if a bug is difficult to debug and fix, its report would be open for a long time, developers tend to discuss it more, and its patch could cover more lines of code changes.

Table 4.2 reports our measurement results. We observe that on average, it takes developers about two months to debug and fix a performance bug in an Android application. During this period, they can have tens of rounds of discussions, resulting in many bug comments (up to 71). Besides, on average, bug-fixing patches can cover more than 182 lines of code changes, indicating non-trivial bug-fixing effort. For comparison, we also randomly selected 200 non-performance bugs (bugs without performance labels) from the bug database of Firefox and Chrome (we selected 100 bugs for each). We did not select non-performance bugs from other application subjects for comparison, because each of these subjects contains only a few performance bugs (about two to four). Such small sample sizes may lead to unreliable comparison results, leaving a weak foundation for further research on related topics [47]. On the other hand, the vast majority of our studied performance bugs come from Firefox and Chrome, and therefore we selected non-performance bugs from these two subjects for comparison. The severity levels of our selected 200 non-performance bugs are comparable to those of performance bugs in Firefox and Chrome. Figure 4.2 compares these two kinds of bugs by boxplots ("Perf" means "Performance bug" and "NPerf" means "non-performance bug" in the figure). The results consistently show that performance debugging and bug-fixing require more effort than their non-performance counterparts. For example, in Firefox, the median bug open duration is

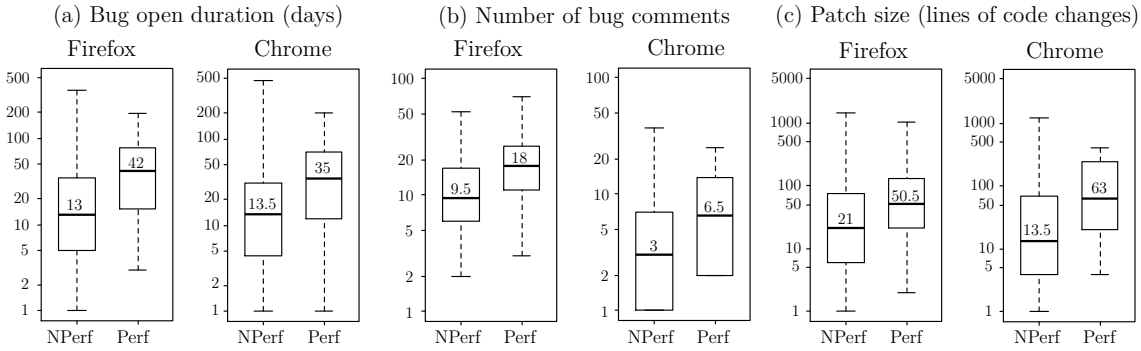

Figure 4.2: Comparison of debugging and bug-fixing effort

Table 4.3: p-values of Mann-Whitney U-tests

Subject	***p*-value**		
	Bug open duration	**# bug comments**	**Patch size**
Firefox	0.0008	0.0002	0.0206
Chrome	0.0378	0.0186	0.0119

42 workdays for performance bugs, but only 13 workdays for non-performance bugs. To understand the significance of the differences between these two kinds of bugs, we conducted a Mann-Whitney U-test [71] with the following three null hypotheses:

- Performance debugging and bug-fixing do not take a significantly longer time than their non-performance counterparts.
- Performance debugging and bug-fixing do not need significantly more discussion than their non-performance counterparts.
- Patches for fixing performance bugs are not significantly larger than those for fixing non-performance bugs.

Table 4.3 gives our Mann-Whitney U-test results (p-values). The results rejected the above three null hypotheses all with a confidence level over 0.95 (i.e., p-values are all less than 0.05). Thus, we conclude that debugging and fixing performance bugs indeed requires more effort than debugging and fixing non-performance bugs. This result can help developers better understand and prioritize bugs for fixing in a cost-effective way, as well as estimating possible manual effort required for fixing certain bugs.

We further looked into bug comments and bug-fixing patches to understand: (1) why it is difficult to debug and fix performance bugs in Android applications, and (2) what support is expected in debugging and bug-fixing. We found that quite a few (22 / 70 = 31.4%) performance bugs involve multiple threads or processes, which may have complicated the debugging and bug-fixing tasks. In addition, these performance bugs rarely caused fail-stop consequences such as application crashes. Due to this reason, traditional debugging information (e.g., stack trace) can offer little help in performance debugging. We analyzed all 70 performance bugs, and found that only four bugs have had their debugging and fixing tasks receiving some help from such traditional information, as judged from their bug discussions (e.g., c:geo bug 949 [6] and Firefox bug 721216 [15]). On the other hand, we found that debugging information from two kinds of tools has received more attention:

Profiling tools. Profiling tools (or profilers) monitor an application's execution, record its runtime information (e.g., execution time of a code unit), and trace details of its resource consumption (e.g., memory). For example, Firefox and Chrome developers often take three steps in performance debugging: (1) reproducing a performance bug with the information provided in its bug report if any, (2) running the application of concern for a long while to generate a profile using their own profilers [8, 13], and (3) performing offline profile analysis to identify performance bottlenecks/bugs if possible. However, profile analysis can be very time-consuming and painful, because current tools (e.g., those from Android SDK) can record tons of runtime information, but which runtime information can actually help performance debugging is still an open question. Firefox developers have designed some visualization tools (e.g., Cleopatra [13]) to save manual effort in profile analysis, but these tools are not accessible to other developers or applicable to other applications. Researchers and practitioners are thus encouraged to design new general techniques and tools for analyzing, aggregating, simplifying and visualizing profiling data to facilitate performance debugging.

Performance measurement tools. Performance measurement tools can also ease performance debugging. They can directly report performance for a selected code unit in an application. For example, Firefox's frame rate meter [14] measures

the number of frames a graphics-rendering unit can produce per second (e.g., when debugging Firefox bug 670930 [15]). This information can help developers in two ways. First, it prioritizes the code units that need performance optimization. Second, it suggests whether a code unit has been adequately optimized. For example, Firefox developers could stop further optimizing a graphics-rendering unit if the frame rate meter reports a score of 60 frames per second (e.g., when fixing Firefox bug 767980). Chrome developers also use similar tools (e.g., using smoothness measurement tools for debugging Chrome bug 242976 [7]). Such tools are useful and welcomed by Android developers. We show some comments about Firefox's frame rate meter from the developers' mailing list:

> *"I found it very useful for finding performance issues in Firefox UI, and web devs should find it useful too."*

> *"This is fantastic stuff. It's a must-have for people hacking on front end UI. Also for devs tracking animation perf."*

Besides understanding the challenges of performance debugging, we also looked for reasons from bug-fixing patches why fixing performance bugs is so difficult. We found that such patches are often complex and have to conduct: (1) algorithmic changes (e.g., Firefox bug 767980 [15]), (2) design pattern reimplementation (e.g., Firefox bug 735636 [15]), or (3) data structure or caching scheme redesign (e.g., Chrome bug 245782 [7]). Such bug-fixing tasks are usually complex. This explains why fixing performance bugs took a longer time and incurred much larger patch sizes than fixing non-performance bugs, as illustrated in Figure 4.2.

Based on these findings, we answer our research question RQ3: *Debugging and fixing performance bugs are generally more difficult than debugging and fixing non-performance bugs. Information provided by profilers and performance measurement tools are more helpful for debugging than traditional information like stack trace. Existing profilers expect improvement on automatically analyzing, aggregating, simplifying and visualizing collected runtime profiles.*

```
1.     public void refreshThumbnails() {
2.       //generate a thumbnail for each browser tab
3.   -   Iterator<Tab> iter = tabs.values().iterator();
4.   -   while (iter.hasNext())
5.   -     GeckoApp.mAppContext.genThumbnailForTab(iter.next());
6.   +   GeckoAppShell.getHandler().post(new Runnable() {
7.   +     public void run() {
8.   +       Iterator<Tab> iter = tabs.values().iterator();
9.   +       while (iter.hasNext())
10.  +         GeckoApp.mAppContext.genThumbnailForTab(iter.next());
11.  +     }
12.  +   });
13.    }
```

Note: the method genThumbnailForTab() compresses a bitmap to produce a thumbnail for a browser tab.

Figure 4.3: Firefox bug 721216

4.1.5 Common Patterns of Performance Bugs

To learn the root causes of our 70 performance bugs, we studied their bug reports, patches, commit logs and patch reviews. We managed to figure out root causes for 52 of these bugs. For the remaining 18 bugs, we failed due to the lack of informative materials (e.g., related bug discussions).

Performance bugs in Android applications can have complex or application-specific root causes. For example, Firefox's "slow tab closing" bug was caused by heavy message communications between its native code and Java code (Firefox bug 719494 [15]), while AnkiDroid suffered GUI lagging because its database library was inefficient (AnkiDroid bug 876 [5]). Despite such variety, we still identified three common causes for 21 out of the 52 performance bugs (40.4%). We explain them with concrete examples below.

Lengthy operations in main threads. As mentioned in Section 2.2, Android applications should not block their main threads with heavy tasks [1]. However, when applications become increasingly more complex, developers tend to leave lengthy operations in main threads. We observed quite a few occurrences of such bugs (11 / 52 = 21.2%). Figure 4.3 gives a simplified version of Firefox bug 721216 [15] and its bug-fixing patch. This bug caused Firefox to suffer from GUI lagging when its "tab strip" button was clicked. The bug occurred because the button's

```
1.    public class ZmanimActivity extends Activity {
2.      private ZmanimLocationManager lm;
3.      private ZmanimLocationManager.Listener locListener;
4.      public void onCreate() {
5.        //get a reference to system location manager
6.        lm = new ZmanimLocationManager(ZmanimActivity.this);
7.        locListener = new ZmanimLocationManager.Listener() {
8.          public void onLocationChanged(ZmanimLocation newLoc) {
9.            //build UI using obtained location in a new thread
10.           rebuildUI(newLoc);
11.         }
12.       };
13.       //register location listener
14.       lm.requestLocationUpdates(GPS, 0, 0, locListener);
15.     }
16.     public void onResume() {
17. +     //register location listener if UI still needs update
18. +     if(buildingUINotFinished)
19. +       lm.requestLocationUpdates(GPS, 0, 0, locListener);
20.     }
21.     public void onPause() {
22. +     //unregister location listener
23. +     lm.removeListener(locListener);
24.     }
25.     public void onDestory() {
26. -     //unregister location listener
27. -     lm.removeListener(locListener);
28.     }
29.   }
```

Figure 4.4: Zmanim bug 50

click event handler transitively called a `refreshThumbnails()` method, which produced a thumbnail for each browser tab by iteratively calling heavy-weight Bitmap compression APIs (Lines 3–5). Later to fix this bug, developers moved such heavy operations to a background thread (Lines 6–12), which can asynchronously update Firefox's GUI.

Wasted computation for invisible GUI. When an Android application switches to background, it may still keep updating its invisible GUI. This brings almost no perceptible benefit to its user, and thus the performed computation (e.g., collecting data and updating GUI) simply wastes resources (e.g., battery power). Such bugs also form a common pattern, which covers 6 of the 52 performance bugs (6 / 52 = 11.5%). For instance, Figure 4.4 lists the concerned code and corresponding bug-

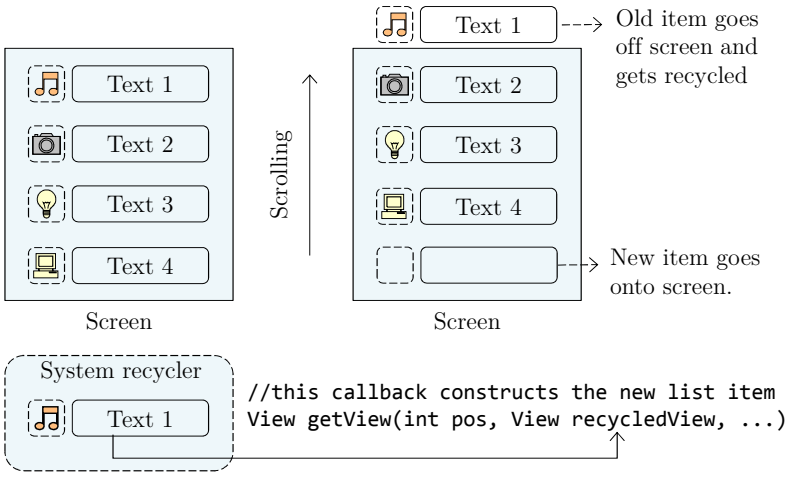

Figure 4.5: List view example

fixing patch for our aforementioned energy leak in Zmanim (bug 50) [37]. When `ZmanimActivity` launches, it registers a location listener to receive location changes for updating its GUI (Lines 5–14). The location listener is normally unregistered when the activity is destroyed (Line 27). However, if a user launches Zmanim and then switches it to background (Android OS will call `onPause()` and `onStop()` handlers accordingly, but not `onDestroy()`), the application will keep receiving location changes to update its GUI, which is, however, invisible. The location sensing and GUI refreshing are then useless, but still drain battery power. This can be common for many smartphone applications, because users often perform multiple tasks at the same time (e.g., playing Facebook and Twitter while listening to music) and frequently switch between them. To fix such bugs, developers have to carefully monitor application states and disable unnecessary tasks when an application goes to background. For example, Firefox developers suggested disabling timers, animations, DOM events, audio, video, flash plugins, and sensors when Firefox went to background (Firefox bug 736311 [15]). Similarly, as Figure 4.4 shows, Zmanim developers disabled location sensing by unregistering the location listener in `ZmanimActivity`'s `onPause()` handler (Line 23), and enabled it again in `onResume()` handler when necessary (Lines 17–19).

Frequently invoked heavy-weight callbacks. Four out of the 52 performance bugs (4 / 52 = 7.7%) concern frequently invoked callbacks. These callbacks need to be light-weight since they are frequently invoked by Android OS. However, many

such callbacks in real-world applications are ill-implemented. They are heavy-weight and can significantly slow down concerned applications. We illustrate such bugs with a list view callback example below.

A list view displays a list of scrollable items and is widely used in Android applications. Figure 4.5 gives one example, where each listed item contains two elements (i.e., two inner views of the list item): an icon and a text label. When a user scrolls up a list view, some existing items will go off the top of the screen while some new items will be added to the bottom. To use a list view, developers need to write an adapter class and define its `getView()` callback (see Figure 4.6 for example). At runtime, when a new item needs to go onto the screen, Android OS will invoke the `getView()` callback to construct and show this item. This callback conducts two operations: (1) parsing the new item's layout XML file and constructing a tree of its elements (a.k.a., list item layout inflation), and (2) traversing the tree to retrieve specific elements for updating (a.k.a., inner view retrieval and update). However, XML parsing and tree traversing can be time-consuming when a list item's layout is complex (e.g., containing many elements or having hierarchical structures as Android applications typically do). Screen scrolling can thus slow down if such operations are commonly performed. For performance concerns, Android OS recycles each item that goes off the screen while users scroll a list view. The recycled items can be reused to construct new items that need to appear later. Such "recycle and reuse" can be done as list items often have identical layouts.

We give two versions of `getView()` implementation in Figure 4.6. The first inefficient version conducts two aforementioned operations (Lines 2–9) every time the callback is invoked. The second version applies a "view holder" design pattern suggested by Android documentation [67]. The basic idea is to reuse previously recycled list items. It avoids list item layout inflation when there are recycled items for reuse (Lines 24–25). Besides, when a list item is constructed for the first time, the references to its inner view objects are identified and stored in a special data structure (Lines 18–22; data structure defined at Lines 32–36). Later, when reusing recycled items, these stored references can be used directly for updating content (Lines 27–29), avoiding inner view retrieval operations. By doing so, the view holder

```
    //inefficient version
 1. public View getView(int pos, View recycledView, ViewGroup parent) {
 2.   //list item layout inflation
 3.   View item = mInflater.inflate(R.layout.listItem, null);
 4.   //find inner views
 5.   TextView txtView = (TextView) item.findViewById(R.id.text);
 6.   ImageView imgView = (ImageView) item.findViewById(R.id.icon);
 7.   //update inner views
 8.   txtView.setText(DATA[pos]);
 9.   imgView.setImageBitmap((pos % 2) == 1 ? mIcon1 : mIcon2);
10.   return item;
11. }

12. //apply view holder pattern
13. public View getView(int pos, View recycledView, ViewGroup parent) {
14.   ViewHolder holder;
15.   if(recycledView == null) { //no recycled view to reuse
16.     //list item layout inflation
17.     recycledView = mInflater.inflate(R.layout.listItem, null);
18.     holder = new ViewHolder();
19.     //find inner views and cache their references
20.     holder.text = (TextView) recycledView.findViewById(R.id.text);
21.     holder.icon = (ImageView) recycledView.findViewById(R.id.icon);
22.     recycledView.setTag(holder);
23.   } else {
24.     //reuse the recycled view, retrieve the inner view references
25.     holder = (ViewHolder) recycledView.getTag();
26.   }
27.   //update inner view contents
28.   holder.text.setText(DATA[pos]);
29.   holder.icon.setImageBitmap((pos % 2) == 1 ? mIcon1 : mIcon2);
30.   return recycledView;
31. }

32. //view holder class for caching inner view references
33. public class ViewHolder {
34.   TextView text;
35.   ImageView icon;
36. }
```

Figure 4.6: View holder pattern

pattern can save both computation for list item layout inflation and inner view retrieval, and memory for constructing new list items. Frequently invoked callbacks should adopt such efficient designs.

Based on these findings, we answer our research question RQ4: *Our study identified three common performance bug patterns: (1) lengthy operations in main threads, (2) wasted computation for invisible GUI, and (3) frequently-invoked heavy-weight callbacks. Researchers and practitioners should design effective techniques to prevent and detect such performance bugs.*

4.1.6 Discussions

Our findings of performance bugs in smartphone applications exhibit some unique features, as compared with those from PC or server-side applications.

- Smartphone application platforms are new and quickly evolving. For example, the current Android platform is not fully optimized and developers keep improving its performance for better user experience. Smartphone applications are thus susceptible to performance bugs due to such platform instability. As shown earlier, Android users can easily manifest performance bugs by simple daily operations.

- Smartphone applications run on devices with small-sized touch screens. Users interact with these devices in a way that is very different from what they do with PCs. For example, users perform screen scrolling much more frequently on smartphones than on PCs. This makes GUI responsiveness and smoothness more crucial for smartphones. Our study reported that GUI-related bugs are pervasive and have become a dominant bug type that concerns Android applications' performance.

- Smartphone applications run on devices with small-capacity batteries, but can access energy-consuming components like GPS sensor and accelerometer, which are usually not available on PCs. Cost-ineffective uses of such components can lead to high energy consumption. Indeed, we found energy leaks

were clearly severe in our studied Android applications. These comparisons help researchers and practitioners understand how performance bugs occur in smartphone applications, as well as explaining why they differ from their counterparts in traditional PC or server-side applications.

Threats to validity. The validity of our study results may be subject to several threats. The first is the representativeness of our selected Android applications. To minimize this threat, we selected eight large-scale and popularly-downloaded Android applications, which cover five different categories. We wish to generalize our findings to more applications, and we will show in Section 4.3 how our findings can help detect performance bugs for a wider range of Android applications. The second threat is our manual inspection of the selected performance bugs. We understand that manual inspection can be error-prone. To reduce this threat, we asked participants to conduct manual inspections independently. We also re-examined and cross-validated all results for consistency.

Algorithm 1: Detecting lengthy operations in main thread

Input : Application classes *Classes*
A list of heavy-weight or blocking APIs *APIs*

Output: A set of detected warnings *Report*

```
1.  foreach cls ∈ Classes do
2.      checkpoints ← ∅;
        // analyze class hierarchy
3.      if isAppComponent(cls) then
4.          handlers ← findLifecycleHandlers(cls);
5.          checkpoints.add(handlers);
6.      end
7.      if isGUIEventListener(cls) then
8.          handler ← getGUIEventHandler(cls);
9.          checkpoints.add(handler);
10.     end
11.     if checkpoints ≠ ∅ then
12.         foreach checkpoint ∈ checkpoints do
13.             callgraph ← makeCallGraph(checkpoint);
                // depth-first call graph traversal
14.             stack.push(checkpoint);
15.             while stack is not empty do
16.                 method ← stack.pop();
17.                 if method not visited before then
18.                     label method as visited;
19.                     if method ∈ APIs then
20.                         add a warning to Report;
21.                     end
22.                     push all callees of method to stack;
23.                 end
24.             end
25.         end
26.     end
27. end
```

Algorithm 2: Detecting violations of the view holder pattern

Input : Application classes $Classes$

Output: A set of detected warnings $Report$

```
1.  foreach cls ∈ Classes do
        // analyze class hierarchy
2.      if isListViewAdapter(cls) then
3.          callback ← findGetViewCallback(cls);
4.          cfg ← makeControlFlowGraph(callback);
5.          pdg ← makeProgramDependencyGraph(cfg);
            // traverse the PDG to find interesting CFG nodes
6.          recycledItemCheckingNodes ←
            findRecycledItemCheckingNodes(pdg);
7.          viewInflationNodes ← findViewInflationNodes(pdg);
8.          viewRetrievalNodes ← findviewRetrievalnNodes(pdg);
9.          if recycledItemCheckingNodes = ∅ then
                // recycled list item is not reused
10.             add a warning to Report;
11.         else
12.             foreach viNode ∈ viewInflationNodes do
13.                 depend ←
                    checkDependency(pdg, viNode, recycledItemCheckingNodes);

14.                 if depend ≠ true then
                        // unconditional list item layout inflation
15.                     add a waning to Report;
16.                 end
17.             end
18.             foreach vrNode ∈ viewRetrievalNodes do
19.                 depend ←
                    checkDependency(pdg, vrNode, recycledItemCheckingNodes);

20.                 if depend ≠ true then
                        // unconditional inner view retrieval
21.                     add a warning to Report;
22.                 end
23.             end
24.         end
25.     end
26. end
```

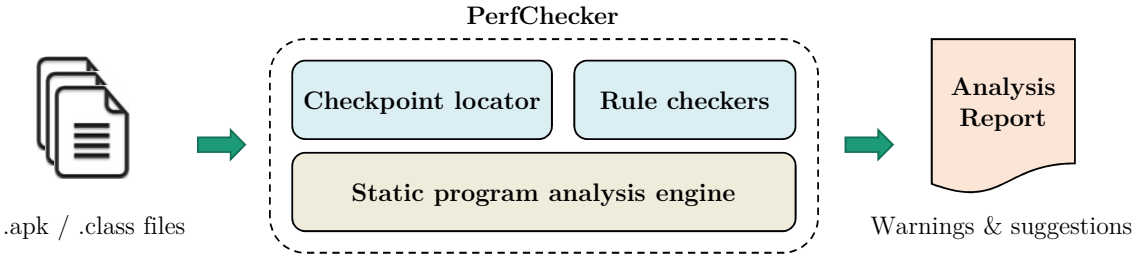

Figure 4.7: Overview of our static analysis technique

4.2 Rule-based Performance Bug Detection

In order to help developers locate performance bugs in their applications, we designed a light-weight static performance analysis tool PerfChecker on top of Soot [33], a widely-used static analysis framework for Java programs, after observing common performance bug patterns. Figure 4.7 gives a high level overview of PerfChecker. It takes as input an Android application's Java bytecode, performs static code analysis, and generates warnings if it detects any issues that may affect the performance of the application under analysis.[3] Our current version of PerfChecker supports detecting two of our identified performance bug patterns: (1) lengthy operations in main threads, and (2) violations of the view holder pattern (as a concrete case of the "frequently invoked heavy-weight callbacks" bug pattern). In the following, we elaborate on the bug detection algorithms.

Detecting lengthy operations in main threads. Algorithm 1 describe the detection process. For each application class, PerfChecker first conducts a class hierarchy analysis to identify a set of checkpoints (Lines 2–10). These checkpoints include the lifecycle event handlers defined in application component classes (e.g., the `onCreate()` handler of those classes that extend the `Activity` class) and GUI event handlers defined in GUI event listener classes (e.g., the `onClick()` handler of those classes that implement the `View.OnClickListener` interface). According to Android's single thread policy, these checkpoints are all executed in an Android application's main thread [1]. Therefore, any methods that are transitively called

[3]PerfChecker can also take an application's installation .apk file as input and transform the Dalvik bytecode to Java bytecode for analysis.

by a checkpoint would by default run in an application's main thread unless they are explicitly put to run in worker threads, which can be created by using various concurrency programming mechanisms such as the APIs defined in the `AsyncTask` class. Then, to detect lengthy operations in an Android application's main thread, PerfChecker constructs a call graph for each checkpoint, and traverses the graph in a depth-first manner to check whether the following rule holds: the checkpoint should not transitively invoke any heavy-weight or blocking operations (Lines 12–25). Currently, in our implementation, we have considered the following well-known heavy-weight or blocking operations: networking, database query, file IO, Bitmap resizing and media decoding. PerfChecker would check if any APIs for conducting these operations are transitively invoked by any checkpoint. If yes, PerfChecker would report warnings accordingly.

Detecting violations of the view holder pattern. Algorithm 2 describes the view holder pattern violation detection process. Similarly, for each application class, PerfChecker first conducts a class hierarchy analysis to identify a set of checkpoints including all `getView()` callbacks defined in list view adapter classes (Lines 2–3). PerfChecker then constructs a program dependency graph for each checkpoint (Lines 4–5), and traverses the graph to check whether the following three rules are violated:

- **List item reuse.** The list view callback should check if there are reusable list items that were recycled by the system (Lines 9–10).
- **Conditional list item layout inflation.** List item layout inflation operations should be conditionally conducted based on whether there are reusable list items (Lines 12–17).
- **Conditional inner view retrieval.** Inner view retrieval operations should be conditionally conducted based on whether there are reusable list items (Lines 18–23).

PerfChecker would report a warning if it finds any violations of the above rules.

4.3 Experimental Evaluation

In this section, we conduct experiments to investigate whether our static performance analysis tool can help developers fight with performance bugs in real-world Android applications. The evaluation is driven by the following two research questions:

- **RQ5 (Effectiveness and efficiency):** *Can PerfChecker efficiently and effectively identify performance optimization opportunities in real-world Android applications?*
- **RQ6 (Performance improvement):** *How much can we improve the performance of an Android application if its performance bugs detected by PerfChecker are fixed?*

4.3.1 Effectiveness and Efficiency of PerfChecker

A. Experimental setup

To answer research question RQ5 about the effectiveness and efficiency of our static performance analysis technique, we conducted experiments on real-world Android applications. For the experiments, we selected two sets of Android applications as subjects. The first set consists of the 29 open-source applications discussed earlier in our empirical study (see Section 4.1). The second set consists of 10 top free commercial Android applications selected from Google Play store. Tables 4.4 and 4.5 give the details of these applications. The tables report for each application: (1) its name, (2) its category, (3) the revision / version used in our experiment,[4] (4) its size, (5) its number of downloads, and (6) its source code availability if it is open-source. As we can see, our selected applications are large-scale, diverse (e.g., covering 14 different categories) and popular on market. We then downloaded these applications' source code (for open-source applications) or installation .apk file (for commercial

[4]We chose the latest version / revision of these applications as our experimental subjects.

Table 4.4: Open-source Android applications used in PerfChecker evaluation

Application name	Application Category	Revision no.	Size (LOC)	Downloads	Availability
Ushahidi	Communication	59fbb533d0	43.3K	10K – 50K	GitHub
c:geo	Entertainment	6e4a8d4ba8	37.7K	1M – 5M	GitHub
Omnidroid	Productivity	865	12.4K	1K – 5K	Google Code
Open GPS Tracker	Travel & Local	14ef48c15d	18.1K	100K – 500K	Google Code
Geohash Droid	Entertainment	65bfe32755	7.0K	10K – 50K	Google Code
Android Wifi Tether	Communication	570	9.2K	1M – 5M	Google Code
Osmand	Travel & Local	8a25c617b1	77.4K	500K – 1M	Google Code
My Tracks	Health & Fitness	e6b9c6652f	27.1K	10M – 50M	Google Code
WebSMS	Communication	1f596fbd29	7.9K	100K – 500K	Google Code
XBMC Remote	Media & Video	594e4e5c98	53.3K	1M – 5M	Google Code
ConnectBot	Communication	716cdaa484	33.7K	1M – 5M	Google Code
Firefox	Communication	895a9905dd	122.9K	10M – 50M	Mozilla Repositories
APG	Communication	a6a371024b	98.2K	50K – 100K	Google Code
FBReaderJ	Books & References	0f02d4e923	103.4K	5M – 10M	GitHub
Bitcoin Wallet	Finance	12ca4c71ac	35.1K	100K – 500K	Google Code
AnySoftKeyboard	Tools	04bf623ec1	26.0K	500K – 1M	GitHub
OI File Manager	Productivity	f513b4d0b6	7.8K	5M – 10M	GitHub
IMSDroid	Media & Video	553	21.9K	100K – 500K	Google Code
Chrome	Communication	58e8ec4010	77.3K	50M – 100M	Google Code
EbookDroid	Productivity	2013	69.8K	1M – 5M	Google Code
Remote Notifier	Productivity	7c31ec8497b5	6.1K	100K – 500K	Google Code
SMSDroid	Communication	f5f2d9285e	6.9K	100K – 500K	Google Code
Zmanim	Books & References	505	4.3K	10K – 50K	Google Code
Barcode Scanner	Shopping	2833	10.1K	100M – 500M	Google Code
Osmdroid	Travel & Local	863	10.3K	50K – 100K	Google Code
AnkiDroid	Education	3f1522c2ab	44.8K	500K – 1M	Google Code
Sofia Public Transport Nav.	Transportation	d3360500cdc4	2.5K	50K – 100K	Google Code
CSipSimple	Communication	2278	46.6K	1M – 5M	Google Code
K-9 Mail	Communication	57c998becc	76.2K	1M – 5M	Google Code

Table 4.5: Commercial Android applications used in PerfChecker evaluation

Application name	Category	Version	Size (MB)	Downloads
Reddit is fun	News & Magazines	3.1.13	4.0	1M – 5M
WeChat	Communication	5.1	25.5	100M – 500M
BBC News	News & Magazines	2.5.2	2.3	5M – 10M
Sina Weibo	Social	4.2.6	22.4	5M – 10M
Flipboard	News & Magazines	2.2.7	5.2	100M – 500M
Facebook	Social	6.0.0.28.28	14.9	500M – 1B
LINE	Communication	4.0.1	19.4	100M – 500M
Skype	Communication	4.6.0.42007	17.2	100M – 500M
Dropbox	Productivity	2.3.12.10	15.1	100M – 500M
Twitter	Social	5.2.2	9.0	100M – 500M

Notes: 1M=1,000,000 and 1B= 1,000,000,000. We only count downloads from Google Play store.

applications) from corresponding software hosting platforms or Google Play store. For open-source applications, we complied them for their target Android platform version and obtained the Java bytecode for analysis. For commercial applications, whose source code are not available, we transformed the Dalvik bytecode in their .apk file to Java bytecode using the dex2jar tool [11]. After such preparation, we then applied our tool on these application subjects for experiments. All our experiments were conducted on a Linux server with 16 cores of Intel Xeon CPU @2.10GHz running CentOS 6.4 64-bit. When analyzing each application subject, we set the maximum JVM heap size to 4 GB. In the following, we discuss the results of our experiments on the two sets of Android applications.

B. Experiment results on open-source Android applications

In our first set of experiments, we applied PerfChecker to the 29 open-source Android applications, which comprise more than 1.1 million lines of Java code, to study its effectiveness and efficiency of detecting performance bugs. PerfChecker quickly finished the analyses of all subjects. Table 4.6 gives its analysis time for

Table 4.6: Analysis time for open-source Android applications

Application name	Analysis time (seconds)	
	VH checker	LM checker
Ushahidi	3.29	35.94
c:geo	2.21	32.07
Omnidroid	2.82	8.75
Open GPS Tracker	1.83	15.68
Geohash Droid	1.17	6.35
Android Wifi Tether	1.41	9.82
Osmand	3.97	19.76
My Tracks	2.11	24.33
WebSMS	1.18	14.24
XBMC Remote	3.32	17.69
ConnectBot	2.01	16.89
Firefox	7.63	276.68
APG	2.21	10.72
FBReaderJ	2.64	23.89
Bitcoin Wallet	2.35	20.51
AnySoftKeyboard	1.65	7.22
OI File Manager	1.69	10.23
IMSDroid	2.20	15.12
Chrome	6.31	112.24
EbookDroid	3.67	26.72
Remote Notifier	2.97	9.72
SMSDroid	2.26	15.74
Zmanim	5.58	12.42
Barcode Scanner	8.71	13.26
Osmdroid	1.39	9.97
AnkiDroid	3.97	46.79
Sofia Public Transport Nav.	1.46	5.85
CSipSimple	3.97	17.18
K-9 Mail	6.12	61.64

Table 4.7: Performance bugs detected in open-source Android applications

Application name	Reported warnings		True bug pattern instances		Bug ID(s)
	VH (86)	LM (82)	VH (69)	LM (57)	
Ushahidi	13	3	**9***	**2**	146, 159
c:geo	7	8	0	**5**	3054
Omnidroid	9	8	9	8	182, 183
Open GPS Tracker	3	1	1	0	390
Geohash Droid	0	1	0	1	48
Android Wifi Tether	1	3	1	3	1829, 1856
Osmand	19	19	**18**	**17**	1977, 2025
My Tracks	2	5	**2***	0	1327
WebSMS	0	1	0	**1**	801
XBMC Remote	3	3	1	0	714
ConnectBot	0	7	0	6	658
Firefox	1	0	**1**	0	899416
APG	4	9	4	8	140, 144
FBReaderJ	6	12	**6***	6	148, 151
Bitcoin Wallet	5	1	**4**	0	190
AnySoftKeyboard	2	0	**2***	0	190
OI File Manager	1	0	**1***	0	39
IMSDroid	10	1	10	0	457

Note: "VH" means "Violation of the view Holder pattern", and "LM" means "Lengthy operations in Main threads".

each application subject. As we can see, PerfChecker can finish detecting violations of view holder pattern for each application within a few seconds (the second column of Table 4.6), and lengthy operations in main threads within a few minutes (the third column of Table 4.6).[5] The memory consumption of the analyses was not high. During the experiments, we did not observe any memory pressure when the maximum JVM heap size was set to 4 GB. In fact, the analyses of the view holder pattern violations for our application subjects never consumed more than 1 GB memory. The analyses of lengthy operations in main threads can consume maximum 3 GB memory for some large-size application subjects such as Firefox, but still such overhead can be well supported by PCs nowadays. Therefore, we consider our analysis technique efficient and light-weight.

As for the effectiveness, PerfChecker reported 86 warnings of view holder pattern violations and 82 warnings of lengthy operations in main threads in 18 out of the 29 analyzed applications, as shown in Table 4.7.[6] Since the source code of these applications are readily available, we then carefully examined the corresponding source files to check whether the reported warnings are true bug pattern instances or false alarms. After the manual inspection, we found that 126 (75%) of the 168 reported warnings are true bugs. For example, Osmand is a popular map and navigation application for Android devices. PerfChecker detected 17 application classes containing event handlers that invoke file IO and database query APIs in Osmand's main thread, and 18 violations of the view holder pattern.

We reported our findings (i.e., detected bugs) as well as optimization suggestions to corresponding application developers. Encouragingly, we received prompt and enthusiastic feedback from them. Altogether, 68 reported bugs (54.0%; bold and underlined in Table 4.7) have been confirmed by developers as real issues that affect application performance. These issues cover 9 of the 18 applications (50.0%). 20

[5]The results were averaged over three different runs.

[6]For each analyzed application class, PerfChecker may generate several warnings for each bug pattern. For example, it may report that several lifecycle event handlers in an activity class can transitively call heavy APIs. In such cases, when we count the number of warnings, we only count once for the class. This also applies to our experiments on commercial Android applications.

of the 68 confirmed issues (29.4%; marked with "*" in Table 4.7) have been fixed in a timely fashion by following our suggestions. Some developers, although not immediately fixing their confirmed issues, promised to optimize their applications according to our suggestions in future releases (e.g., My Tracks bug 1327 [25]). Other reported issues are pending (their concerned applications may not be under active maintenance). In addition to reporting bugs, we also communicated with developers via bug reports and emails, and obtained some interesting findings as discussed below.

First, developers showed great interest in our performance analysis tool. For example, we received the following feedback:

> *"Thanks for reporting this. I'll take a look at it. Just curious, where is this static code checker? Anywhere I can play with it as well? Thanks."* (Ushahidi bug 159 [35])

> *"Thanks for your report. The code is only a year old. That's probably the reason (why it's not well optimized). Your static analyzer sounds really interesting. I wonder if lint can also check this.*[7]*"* (OI File Manager bug 39 [26])

These comments suggest that developers welcome performance analysis tools to help optimize their Android applications. Our PerfChecker is helpful, especially for complex applications. For example, it detected some applications transitively calling heavy-weight APIs in their main threads, and the call chains can last for tens of method calls (e.g., c:geo bug 3054 [6]). Such bugs can easily escape to production and degrade user experience. Unfortunately, there are few industrial-strength tools to support smartphone applications' performance analysis. Thus there is a strong need for effective tools to help smartphone application developers fight with performance bugs.

[7]Lint is a static checker in Android Studio [3], which is an official integrated development environment for Android applications. It can detect performance threats like using getters instead of direct field accesses within a class, but did not support our identified performance bug patterns at our study time. Details can be found at `http://tools.android.com/tips/lint-checks`.

Second, some developers held conservative attitudes toward performance optimization. They concerned much, e.g., (1) whether optimization can bring significant performance gains, (2) whether optimization can be done easily, and (3) whether optimization helps toward an application's market success. They hesitated to conduct performance optimization when the optimization seem to require a lot of effort but bring no immediate benefits. For example, WebSMS and Firefox developers responded as follows:

> *"You are totally right. WebSMS is ported from J2ME and has a very old code base ... If I would write it from scratch, I'd do things differently. I once started refactoring, but gave up in the end. There were other things to do, and the SMS user base is shrinking globally If you want to help, just fork it on GitHub and let me merge your changes. I'd be very thankful."* (WebSMS bug 801 [36])

> *"Thanks for the report! This shouldn't be a big concern; that UI is not a high-volume part. We'll keep this bug open, and I'd accept a patch which improves the code, but it's not a high-priority work item."* (Firefox bug 899416 [15])

Finally, some developers were cautious and willing to conduct code optimization to improve performance or maintainability for their applications. For example, c:geo developers responded:

> *"Such optimizations are 'micro optimizations', and they do not improve the user visible performance. Good developers however will still refactor code into the better version, mostly to make it more readable."* (c:geo bug 222 [6])

They also quickly fixed this reported performance bug. This may explain why c:geo keeps being highly-rated and popularly-downloaded (1M–5M downloads) on the market.

Table 4.8: Analysis time and detected performance bugs in commercial Android applications

Application name	Analysis time (seconds)	Reported warnings	True violations
Reddit is fun	6.55	8	**2**
WeChat	10.69	18	12
BBC News	2.74	3	0
Sina Weibo	9.19	43	10
Flipboard	6.28	18	**10**
Facebook	3.77	1	0
LINE	8.60	5	**2**
Skype	4.94	13	5
Dropbox	4.22	5	1
Twitter	7.64	16	4

C. Experiment results on commercial Android applications

In our second set of experiments, we applied the view holder pattern violation checker in PerfChecker to the 10 popular commercial Android applications shown in Table 4.5 to study whether PerfChecker can help locate performance optimization opportunities.[8] PerfChecker quickly finished the analyses of these application subjects and reported a set of warnings, as shown in Table 4.8. Similar to our first set of experiments, we manually validated each reported warning by checking the concerned application's source code, which is decompiled from Java bytecode, to see if it is a true bug or false alarm. However, due to obfuscation, the transformed Java bytecode is of low quality and the source code obtained via decompilation has poor comprehensibility. Therefore, in our manual validation, we conservatively consid-

[8]We also applied the lengthy operation checker to these application subjects. However, we were not able to manually validate if the reported warnings are true issues due to the lack of source code and heavy obfuscation (note that we need to check the method call chains of certain event handlers). So we did not include the checker in this study.

ered a reported violation as a "true violation" only when we were highly confident that we correctly understood the concerned source code and it is indeed problematic. The last column of Table 4.8 reports the number of true violations after our validation. We observe that PerfChecker found real performance threats for 8 out the 10 applications. We then reported these true violations to corresponding developer teams and received enthusiastic confirmation and acknowledgement (confirmed violations are underlined in Table 4.8):

> *"We really appreciate your valuable comments and suggestions on improving Line. We would like to pass your comments to the relevant departments, where they may be used for future versions of LINE."* (LINE Customer Support)

> *"It feels great to be tested by your tool and it is awesome to connect with bright developers this way. We want to explore more interesting stuff around Android development with you."* (Flipboard Customer Support)

Interestingly, when analyzing Twitter, we found six warnings of view holder pattern violation in the latest version of Samsung Mobile SDK (1.5 Beta1) [31], which is used by Twitter. We also manually examined these warnings and confirmed that they are all true issues. We reported our findings to Samsung developers . They were very interested and generally agreed that improving the performance of this SDK can benefit both application developers and end users.[9]

> *"With such powerful hardware, developers got lazy and started employing bad habits. ... If we have better SDK (e.g., improving their performance), we have happier developers and happier end-users."* (Samsung developer)

Based on the experiment results discussed above, we can see that our performance analysis tool detected quite a few previously-unknown issues that affect performance in a wide range of real-world Android applications as well as a widely-used

[9]For more details, please refer to the Samsung developer forum: `http://developer.samsung.com/forum/board/thread/view.do?boardName=SDK&messageId=256618`.

application development library. This validates the usefulness of our empirical findings.

D. Discussions

Our PerfChecker reported some false alarms in the experiments. For the open-source applications, 42 (25%) of the 168 reported warnings are not true issues. We checked these warnings and found the major reason of such false positives. In order to precisely locate view holder pattern violations and lengthy operations in main threads, our algorithms need precise call graphs and program dependency graphs as input. However, these graphs constructed by Soot may not be precise in practice. For example, without correctly knowing the type of a method call receiver when resolving a virtual call, Soot may not be able to determine the exact called method and would produce a call graph with imprecise edges. For the commercial applications, we observe that our PerfChecker generated more false alarms (we were only confident that 35.4% of the reported warnings are true issues after manual validation). This is because these commercial applications are heavily obfuscated. The Java bytecode obtained via transforming Dalvik bytecode is of low quality (e.g., some type information can not be preserved after the transformation [78]), which seriously affects the analysis precision of Soot and therefore our PerfChecker. However, this will not be a problem in reality. Our PerfChecker is built to help developers conduct performance analysis on their own smartphone applications. In such usage scenarios, application source code is available and PerfChecker can take high-quality Java bytecode as input for analysis. The results of our first set of experiments confirmed that PerfChecker's analysis precision on open-source application subjects is satisfactory.

During our communications with developers, we observed that many of them are interested in our static performance analysis technique. In view of this, we later contacted Android Studio developers and suggested to integrate the view holder vio-

Table 4.9: Subjects and bugs for the performance comparison experiments

Application name	Bug ID	Buggy class	Fixed revision
Ushahidi	146	`com.ushahidi.android.app.adapters.ListMapAdapter`	e1b4c17b33
My Tracks	1327	`com.google.android.apps.mytracks.fragments.ChooseActivityDialogFragment`	f211d192b2
FBReaderJ	148	`org.geometerplus.android.fbreader.TOCActivity`	0b97ef54f4
AnySoftKeyboard	190	`com.anysoftkeyboard.ui.settings.AddOnListPreference`	1325146f71
OI File Manager	39	`org.openintents.filemanager.bookmarks.BookmarkListAdapter`	8c9c4292ca

lation checker into Lint, Android Studio's built-in static checker [3]. The developers took our suggestion and Lint now has included the checker from version 0.5.2.[10]

4.3.2 Performance Improvement Study

A. Experimental setup

To answer research question RQ6 about potential performance improvement if the performance bugs detected by PerfChecker are fixed, we conducted a set of comparison experiments. Since our reported performance bugs in five open-source application subjects have been fixed by the corresponding developers (see Table 4.7), we used these five applications as our experimental subjects. For the experiments, we randomly selected one fixed performance bug for each of the five applications. Table 4.9 lists the basic information of each selected bug, including (1) the name of the corresponding application subject, (2) the bug ID, (3) the buggy class, and (4) the revision in which developers fixed the bug. These bugs are all view holder pattern violations.

To conduct comparison experiments, we prepared two versions for each application subject: one with the selected performance bug (will be referred to the "buggy

[10]See more details at `http://tools.android.com/recent/androidstudio052released`

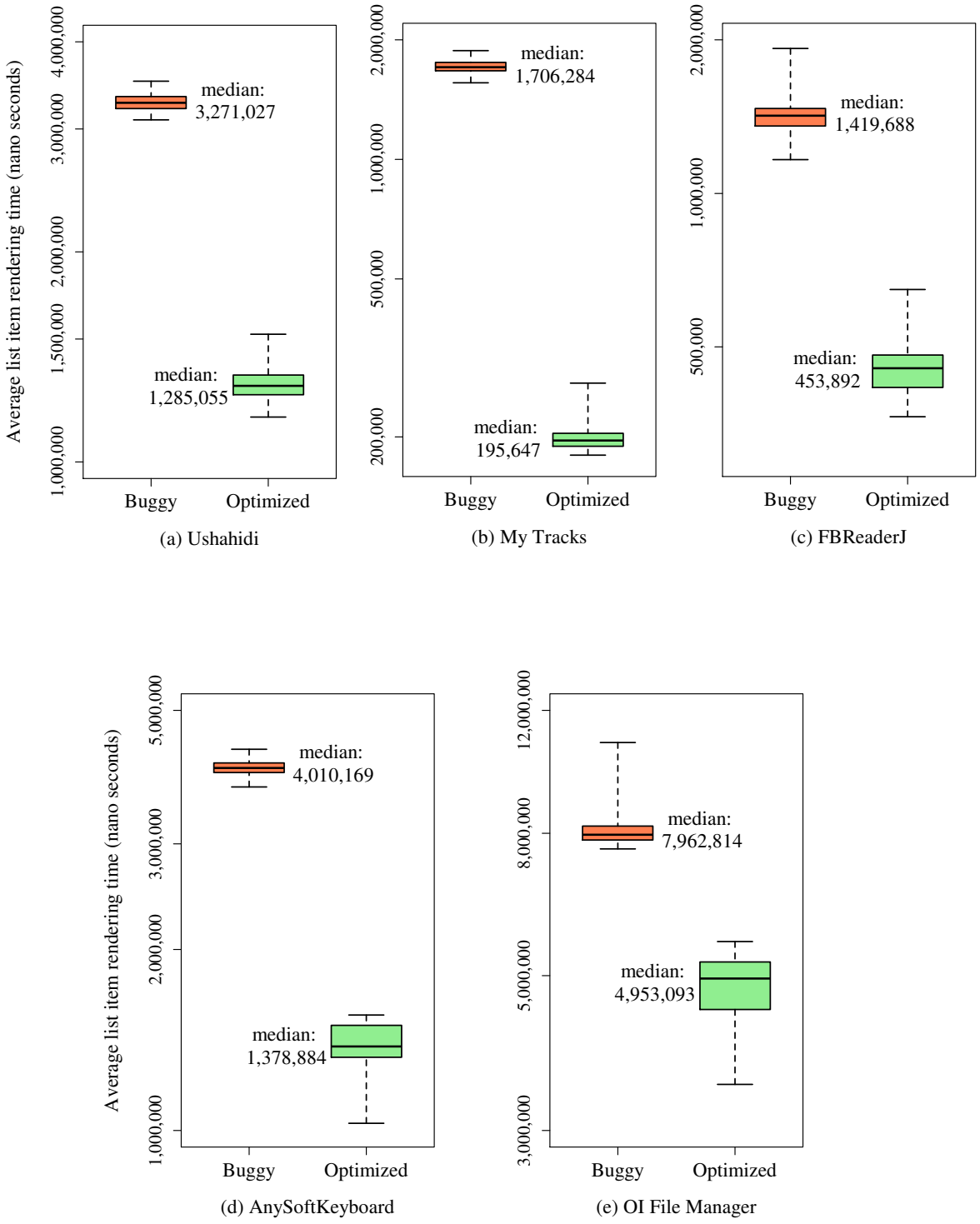

Figure 4.8: Average list item rendering time of studied subjects

version") and the other in which the performance bug has been fixed (will be referred to as the "optimized version"). For the buggy version of each subject, we used the revision, which was used in our previous performance bug detection experiments (see Table 4.4). To prepare the optimized version for each subject, we replaced the buggy version's list view callback `getView()` with that of the bug fixing revision.[11] We did not simply use the bug fixing revision of each subject as its optimized version because there were also other revisions between the buggy version and the bug fixing revision. The code changes in those revisions may have unknown effect on the performance of the application.

Next, we carefully studied our application subjects and designed a test scenario for each of them. In these test scenarios, a tester will interact with the concerned list view widgets in an intuitive manner, which represents real user behaviors. Specifically, each test scenario consists of the following steps: (1) start the application, (2) go to the screen that contains the list view widget (the concrete actions required in this step differ across applications), (3) scroll up the list view to the first item,[12] (4) scroll down the list view to the last item, and (5) exit the application and go to the "Home" screen. To measure the scrolling smoothness of the list view widgets, we manually instrumented the two versions of each subject to calculate and record the time used to render each list item (i.e., the execution time of the `getView()` callback). A list view widget will have better scrolling smoothness if its list items are rendered faster. After these preparations, we then conducted our experiments on a Nexus 5 smartphone running Android 4.4. For each version of an application subject, we ran the corresponding test scenario 100 times using the capture and replay technique [51]. More specifically, before comparing the two versions of each subject, we first manually performed the sequence of actions in the test scenario

[11] In some subjects, we also need to make other necessary modifications besides the callback replacement. For example, the `getView()` callback in the bug fixing revision of FBReaderJ invoked newly defined methods for implementing the view holder pattern and such new methods will also be copied to the buggy version for preparing the optimized version.

[12] The content of list items was prepared by us. For example, in the application OI File Manager, the list view widget is used to display bookmarks to certain folders or files and we created such bookmarks. Such set up is necessary for the comparison experiments.

and captured the interaction using a tool named Finger Replayer [12]. We then used the tool to replay the test scenario on each version 100 times. Besides, we also killed other applications before running each version of our application subject to avoid their effect on the performance of our application subject (e.g., other running applications may compete for computational resources such as CPU and this can affect the list item rendering time of our application subject). To further ensure that we ran the two versions of each application subject in a consistent environment, we always kept the device fully-charged, connected to the power source and running in the airplane mode during the experiments.

B. Experiment results

We now discuss the results of our experiments. For each version of an application subject, we measured its average list item rendering time 100 times. Figure 4.8 gives the results using boxplots. We can see that the results consistently show that the optimized version of each subject requires less time to render list items. Take the application My Tracks for example. The median of the average list item rendering time is 1,706,284 nano seconds for its buggy version and 195,647 nano seconds for its optimized version. In other words, the list items in the optimized version of My Tracks can be rendered over 8 times faster than in the buggy version (the improvements for other application subjects are shown in Table 4.10). To understand whether the optimized version of each application subject can render list items significantly faster than the buggy version, we further conducted Mann-Whitney U-test [71] for each application subject with the following two hypotheses:

- **Null hypothesis** H_0. The average list item rendering time of the optimized version is not significantly smaller than that of the buggy version.
- **Alternative hypothesis** H_1. The average list item rendering time of the optimized version is significantly smaller than that of the buggy version.

Table 4.10 reports the results of the U-tests (p-values). The results rejected the null hypotheses all with high confidence (i.e., p-values are all nearly 0 for all

Table 4.10: Performance improvement and Mann-Whitney U-test results

Application name	Average list item rendering time (nano seconds)			*p*-value of U-test
	Buggy version (mean value)	Optimized version (mean value)	Improvement	
Ushahidi	3,273,540	1,292,408	2.5x	3.6e-41
My Tracks	1,712,820	200,001	8.6x	3.6e-41
FBReaderJ	1,404,155	460,557	3.0x	1.2e-32
AnySoftKeyboard	4,011,937	1,386,251	2.9x	4.3e-36
OI File Manager	8,391,398	4,774,003	1.8x	2.9e-36

application subjects). Thus, we conclude that there is a significant performance improvement for all subjects after the detected bugs are fixed.

In practice, in oder to keep a Android application's user interface (UI) smooth, developers need to make sure that the system can render the UI at a frame rate of 60 FPS (frames per second) or above [1]. To achieve this target, an application typically needs to be able to prepare a list item in a couple of milliseconds. Therefore, even one millisecond saved in rendering a list item is helpful to guarantee satisfactory user experience. This further shows the usefulness of our technique.

4.4 Related Work

Our work in this chapter relates to a large body of existing work on testing, debugging, bug detection and understanding for application performance. We discuss some representative pieces of work in recent years. Some of them focus on smartphone application performance, while others are for PC or server-side applications.

4.4.1 Detecting Performance Bugs

Much research effort has been devoted to automating performance bug/issue detection.[13] For example, Xu et al. used cost-benefit analysis to detect high-cost data structures that bring little benefit to a program's output [96]. Such data structures can cause memory bloat. Xiao et al. used a predictive approach to detect workload-sensitive loops that contain heavy operations, which often cause performance bottlenecks [95]. Recent work Toddler by Nistor et al. detected repetitive computations that have similar memory-access patterns in loops. Such computations can be unnecessary and subject to optimization [77]. These pieces of work focused on performance issues in PC or server-side applications, while there are also other pieces of work particularly focusing on smartphone application performance. For example, Pathak et al. studied no-sleep energy bugs in Android applications and used reaching-definition dataflow analysis to detect such bugs (e.g., an application forgets to unregister a used sensor) [84]. Following in this direction, Guo et al. further proposed a technique to detect general resource leaks, which often cause performance degradation [53]. Similar to Xu et al.'s [96] and Zhang et al.'s work [101], we previously leveraged cost-benefit analysis to detect whether an Android application uses sensory data in a cost-ineffective way (see Chapter 3.2.4). Potential energy leak bugs can be reported after cross-state data utilization comparisons.

4.4.2 Performance Testing

Performance testing is challenging due to the lack of test oracles and effective test input generation techniques. Some ideas have been proposed to alleviate such challenges. For example, Jiang et al. used performance baselines extracted from historical test runs as tentative oracles for new test runs [56]. Grechanik et al. learned rules from existing test runs, e.g., what inputs have led to intensive computations. They used such rules to select new test inputs to expose performance issues [52] These

[13]Some researchers prefer "performance issue" to "performance bug". We do not have a preference and use the two terms interchangeably.

ideas work well for PC applications, but it is unclear whether they are effective for smartphone applications. Our empirical study discloses that many performance bugs in smartphone applications need certain user interaction sequences to manifest. Besides, smartphone applications also have some unique features, e.g., long GUI lagging can force an Android application to close. Such requirements and features should be considered in order to design effective techniques to test the performance of smartphone applications. We have observed initial attempts along this direction. For example, Yang et al. tried to crash an Android application by adding a long delay after each heavy API call to test GUI lagging issues [98]. Jensen et al. studied how to generate user interaction sequences to reach certain targets in an Android application [55]. These attempts support performance testing of Android applications, but how to assess the testing adequacy is still unclear. Our work thus motivates new attempts for performance testing adequacy criteria, as well as effective techniques to expose performance issues in smartphone applications.

4.4.3 Performance Debugging and Optimization

Existing work on debugging and optimization for smartphone application performance mainly falls into two categories. The first category estimates performance for smartphone applications to aid debugging and optimization tasks [54, 63, 83, 102]. For example, Mantis [63] estimated the execution time for Android applications on given inputs. This helps identify problem-inducing inputs that can slow down an application, so that developers can conduct optimization accordingly. PowerTutor [102], Eprof [83] and eLens [54] estimated energy consumption for Android applications by different energy models. They can help debug energy leak issues. For example, eLens offered fine-grained energy consumption estimation at source code level (e.g., method and line level estimation) to help locate energy bottlenecks. The second category of existing work uses profiling to log performance-related information to aid debugging and optimization tasks [87, 89, 100]. For example, ARO [87] monitored cross-layer interactions (e.g., those between the application layer and the resource management layer) to help disclose inefficient resource usage, which commonly causes performance degradation to smartphone applications. Ap-

pInsight [89] instrumented application binaries to identify critical paths (e.g., slow execution paths) in handling user interaction requests, so as to disclose root causes for performance issues. Panappticon [100] shared the same goal as AppInsight, and further revealed performance issues from inefficient platform code or problematic application interactions. There are also performance debugging techniques [58, 91] for PC applications. For example, LagHunter [58] detected user-perceivable latencies in interactive applications (e.g., Eclipse); Shen et al. constructed a system-wide I/O throughput model to guide performance debugging [91]. These techniques may not apply to multi-threaded and asynchronous smartphone applications, because LagHunter tracked only synchronous UI event handling and Shen et al.'s work required a system-level performance model, which may not be available.

4.4.4 Understanding Performance Bugs

Finally, understanding and learning characteristics of performance bugs is a very important step toward designing effective techniques to test and debug performance issues. Existing characteristic studies have mainly focused on PC or server-side applications [57, 76, 99]. For example, Zaman et al. [99] studied performance bug reports from Firefox and Chrome (for PCs), and gave recommendations on how to better conduct bug identification, tracking and fixing. Jin et al. [57] studied the root causes of performance bugs in several selected PC or server-side applications, and identified efficiency rules for their detection. The most recent work by Nistor et al. [76] studied lifecycles of performance bugs (e.g., bug discovery, reporting and fixing), and obtained some interesting findings. For instance, there is little evidence showing that fixing performance bugs has a high chance of introducing new bugs. This encourages developers to conduct performance optimization whenever possible. However, there is a lack of similar studies on performance bugs in smartphone applications. Our work fills this gap by studying 70 real-world performance bugs from large-scale and popularly-downloaded Android applications. Our study also reveals some interesting findings, which differ from those for performance bugs in PC or server-side applications. These findings can help researchers and practitioners to better understand performance bugs in smartphone applications, as well as

proposing new techniques to fight with these bugs (e.g., as we did in our evaluation part).

4.5 Chapter Summary

In this chapter, we conducted an empirical study of 70 performance bugs from real-world Android applications. We reported our study results, which revealed several unique features of performance bugs in smartphone applications. We also identified three common bug patterns, which can support related research on bug detection, performance testing and debugging. To validate the usefulness of our empirical findings, we designed a static performance analysis technique PerfChecker, which supports detecting two of our identified performance bug patterns. We implemented PerfChecker and applied it to a large set of real-world Android applications. It detected many real and previously-unknown performance issues and developers have fixed some critical ones after we reported to them. In addition, we also showed by comparison experiments that fixing the bugs detected by PerfChecker can indeed significantly improve the performance of the corresponding applications. This encouragingly confirmed the usefulness of our empirical findings and performance analysis technique.

Chapter5

Conclusions

In this chapter, we summarize the research work we have completed in this thesis, discuss our ongoing work, and explore our future work.

5.1 Summary of Completed Work

Energy efficiency and performance are two critical factors that affect mobile applications' quality and user experience. However, many real-world mobile applications suffer from serious energy and performance bugs, causing significant user frustrations. In this thesis, we presented our research on understanding and detecting energy and performance bugs in Android applications.

To understand energy and performance bugs, we conducted two large-scale empirical studies of real energy and performance bugs collected from popular Android applications. We carefully studied the characteristics of these bugs such as how they manifest themselves and the difficulties developers encountered when diagnosing these bugs. We also studied the root causes of these bugs and identified several common bug patterns.

To help developers locate energy and performance bugs in their Android applications, we designed two automated techniques based on our identified bug patterns. The first technique GreenDroid is a dynamic analysis technique. It systematically generates user interaction events sequences to execute an Android application and closely monitors the application's runtime behavior to locate energy bugs that arise from the misuse of device sensors and wake locks. The second technique PerfChecker is a static analysis technique. It automatically scans an Android application's bytecode and checks the application's implementation against a set of efficiency rules

formulated from our empirical studies to locate heavy-weight program callbacks and operations that can seriously reduce an application's responsiveness.

We evaluated GreenDroid and PerfChecker by extensive experiments. The experiment results showed that our technique can efficiently and effectively locate real energy and performance bugs in a wide range of popular open-source and commercial Android applications. Fixing the detected bugs can significantly improve the energy efficiency and performance of these applications. This confirms the usefulness of our research.

5.2 Ongoing Work and Future Work

Our techniques presented in this thesis only addressed several common patterns of energy and performance bugs in Android applications. In reality, energy inefficiency and performance degradation in mobile applications can be caused by various reasons. There is much to explore in this research area.

In our ongoing work, we are studying another common pattern of energy bugs. We found that in many Android applications, there are scenarios where an application unnecessarily keeps the device awake for long time using wake locks. This differs from the missing wake lock deactivation pattern studied in Chapter 3 in that the wake locks are eventually released. Diagnosing such wake lock misuse is challenging because it is difficult to judge whether the computation during the wake lock holding time is the critical computation, which should be protected by wake locks (i.e., judging the necessity of wake locks is difficult). To address this challenge, we are investigating a large number of real-world Android applications to study how they use wake locks. Our preliminary finding is that when an Android application holds a wake lock, it is typically using system resources to perform long running computation that can bring users perceptible benefits (e.g., using the audio system to play music in background). Such findings will help us formulate decidable criteria to support detecting unnecessary use of wake locks.

In future, we plan to conduct more investigations of energy and performance bugs in mobile applications (not restricted to the Android platform), aiming to

better understand these bugs and identify more bug patterns. For example, in our energy bug empirical study, we found that a non-negligible proportion (about 16%) of energy bugs were caused by network issues (e.g., energy-inefficient data transmission). We are planning to study these issues to further extend the detection capability of our techniques. We hope that our research together with related work can help improve the quality and user experience of mobile applications, which can potentially benefit millions of users around the world.

List of Publications

Thesis-related publications:

- **Yepang Liu**, Chang Xu, and Shing-Chi Cheung. Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications. *IEEE Software*, 32(1):67–75, Jan 2015.

- **Yepang Liu**, Chang Xu, Shing-Chi Cheung, and Jian Lu. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering*, 40(9):911–940, Sept 2014.

- **Yepang Liu**, Chang Xu, and Shing-Chi Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, Hyderabad, India, June 2014. **ACM SIGSOFT Distinguished Paper Award.**

- **Yepang Liu**, Chang Xu, and Shing-Chi Cheung. Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications*, PERCOM 2013, pages 2–10, San Diego, CA, USA, Mar 2013.

Other publications:

- Wenhua Yang, **Yepang Liu**, Chang Xu, and Shing-Chi Cheung. A Survey on Dependability Improvement Techniques for Pervasive Computing Systems. In *Science China Information Sciences*, 58(5):1–14, May 2015.

- Xiujiang Li, Yanyan Jiang, **Yepang Liu**, Chang Xu, Xiaoxing Ma, and Jian Lu. User Guided Automation for Testing Mobile Apps. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference*, APSEC 2014, pages 27–34, Jeju, Korea, Dec 2014.

- Wenhua Yang, Chang Xu, **Yepang Liu**, Chun Cao, Xiaoxing Ma, and Jian Lu. Verifying Self-adaptive Applications with Uncertainty. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, ASE 2014, pages 199–209, Vasteras, Sweden, Sept 2014.

- Yueqi Li, Shing-Chi Cheung, Xiangyu Zhang, and **Yepang Liu**. Scaling Up Symbolic Analysis by Removing Z-Equivalent States. In *ACM Transactions on Software Engineering and Methodology*, 23(4):1–32, Aug 2014.

- **Yepang Liu**, Chang Xu, Shing-Chi Cheung, and Wenhua Yang. CHECKERDROID: Automated Quality Assurance for Smartphone Applications. In *International Journal of Software and Informatics*, 8(1):21–41, Aug 2014.

- **Yepang Liu**, and Chang Xu. VeriDroid: Automating Android Application Verification. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, MDS 2013, pages 1–6, Beijing, China, Dec 2013.

- Chang Xu, **Yepang Liu**, Shing-Chi Cheung, Chun Cao, and Jian Lu. Towards Context Consistency by Concurrent Checking for Internetware Applications. In *Science China Information Sciences*, 56(8):1–20, Aug 2013.

- **Yepang Liu**, Chang Xu, and Shing-Chi Cheung. AFChecker: Effective Model Checking for Context-Aware Adaptive Applications. In *Journal of Systems and Software*, 86(3):854–867, March 2013.

References

[1] Android developers website. https://developer.android.com/.

[2] Android official website. http://www.android.com/.

[3] Android Studio. https://developer.android.com/sdk/.

[4] AndTweet issue tracker. https://code.google.com/p/andtweet/issues/.

[5] AnkiDroid issue tracker. https://code.google.com/p/ankidroid/issues/.

[6] c:geo issue tracker. https://github.com/cgeo/cgeo/issues.

[7] Chrome issue tracker. https://code.google.com/p/chromium/issues/.

[8] Chrome testing tools. https://sites.google.com/a/chromium.org/dev/developers/testing.

[9] Crawler4j. https://code.google.com/p/crawler4j/.

[10] CSipSimple issue tracker. https://code.google.com/p/csipsimple/issues/.

[11] dex2jar. https://code.google.com/p/dex2jar/.

[12] Finger Replay: a capture and replay tool for Android. https://play.google.com/store/apps/details?id=com.x0.strai.frep.

[13] Firefox built-in profiler for performance analysis. https://developer.mozilla.org/en-US/docs/Performance.

[14] Firefox frame rate meter tool. http://blog.mozilla.org/devtools/tag/framerate-monitor/.

[15] Firefox issue tracker. https://bugzilla.mozilla.org/.

[16] Geohash Droid issue tracker. https://code.google.com/p/geohashdroid/issues/.

[17] GitHub. https://github.com/.

[18] Google Code. https://code.google.com/.

[19] Google Play store. https://play.google.com/.

[20] Google Play Wikipedia page. http://en.wikipedia.org/wiki/Google_Play.

[21] GPSLogger issue tracker. https://code.google.com/p/gpslogger/issues/.

[22] GreenDroid project website. http://sccpu2.cse.ust.hk/greendroid/.

[23] K9Mail issue tracker. https://code.google.com/p/k9mail/issues/.

[24] Mozilla Cross-References. https://mxr.mozilla.org/.

[25] My Tracks issue tracker. https://code.google.com/p/mytracks/issues/.

[26] OI File Manager issue tracker. https://github.com/openintents/filemanager/issues/.

[27] Omnidroid issue tracker. https://code.google.com/p/omnidroid/issues/.

[28] Osmdroid issue tracker. https://code.google.com/p/osmdroid/issues/.

[29] Real APKLeecher. https://code.google.com/p/real-apk-leecher.

[30] Robotium Testing Framework for Android Applications. http://code.google.com/p/robotium/.

[31] Samsung developers website. http://developer.samsung.com/.

[32] Sofia Public Transport Nav. issue tracker. https://code.google.com/p/sofia-public-transport-navigator/issues/.

[33] Soot: a Java Program Optimization Framework. http://sable.github.io/soot/.

[34] SourceForge. http://sourceforge.net/.

[35] Ushahidi issue tracker. https://github.com/ushahidi/Ushahidi_Android/issues.

[36] WebSMS issue tracker. https://code.google.com/p/websmsdroid/issues/.

[37] Zmanim issue tracker. https://code.google.com/p/android-zmanim/issues/.

[38] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, Cary, NC, USA, 2012.

[39] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 143–162, Nashville, TN, USA, 2008.

[40] Tanzirul Azim and Iulian Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 641–660, Indianapolis, Indiana, USA, 2013.

[41] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, London, United Kingdom, 2007.

[42] James Clause and Alessandro Orso. LEAKPOINT: Pinpointing the Causes of Memory Leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 515–524, Cape Town, South Africa, 2010.

[43] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, San Francisco, California, USA, 2010.

[44] Tuan Dao, Indrajeet Singh, and Harsha V. Madhyastha. Tide: A user-centric tool for identifying energy hungrey applications on smartphones. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, ICDCS '15, Columbus, Ohio, USA, June 2015.

[45] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. The CLOSER: Automating Resource Management in Java. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 1–10, Tucson, AZ, USA, 2008.

[46] Mian Dong and Lin Zhong. Sesame: Self-Constructive Energy Modeling for Battery-Powered Mobile Systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, 2011.

[47] Paul D Ellis. *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*. Cambridge University Press, 2010.

[48] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, Vancouver, BC, Canada, 2010.

[49] K. Etessami and T. Wilke. An Until Hierarchy for Temporal Logic. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 108–117, July 1996.

[50] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, Chicago, Illinois, USA, 2011.

[51] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 72–81, San Francisco, CA, USA, 2013.

[52] Mark Grechanik, Chen Fu, and Qing Xie. Automatically Finding Performance Problems with Feedback-directed Learning Software Testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012.

[53] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in Android applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, pages 389–398, Nov 2013.

[54] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101, San Francisco, CA, USA, 2013.

[55] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated Testing with Targeted Event Sequence Generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 67–77, Lugano, Switzerland, 2013.

[56] Zhen Ming Jiang. Automated Analysis of Load Testing Results. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 143–146, Trento, Italy, 2010.

[57] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, Beijing, China, 2012.

[58] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch Me if You Can: Performance Bug Detection in the Wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 155–170, 2011.

[59] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, London, England, UK, 2012.

[60] Hahnsang Kim, Joshua Smith, and Kang G. Shin. Detecting Energy-greedy Anomalies and Mobile Malware Variants. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 239–252, Breckenridge, CO, USA, 2008.

[61] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 56–70, Hyderabad, India, 2008.

[62] Mikkel Baun Kjærgaard, Jakob Langdal, Torben Godsk, and Thomas Toftkjær. EnTracked: Energy-efficient Robust Position Tracking for Mobile Devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 221–234, 2009.

[63] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic Performance Prediction for Smartphone Applications. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC '13, pages 297–308, San Jose, CA, USA, 2013.

[64] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Verifying Android Applications Using Java Pathfinder. Technical report, HKUST-CS-12-03, 2012.

[65] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications*, PerCom '13, pages 2–10, San Diego, CA, USA, March 2013.

[66] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Diagnosing energy efficiency and performance for mobile internetware applications. *IEEE Software*, 32(1):67–75, Jan 2015.

[67] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, Hyderabad, India, June 2014.

[68] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lu. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering*, 40(9):911–940, Sept 2014.

[69] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Wenhua Yang. CHECKERDROID: Automated Quality Assurance for Smartphone Applications. *International Journal of Software and Informatics*, 8(1):21–41, Aug 2014.

[70] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In

Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI '13, pages 57–70, Lombard, IL, USA, 2013.

[71] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, March 1947.

[72] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage Criteria for GUI Testing. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '01, pages 256–267, Vienna, Austria, 2001.

[73] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. *SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

[74] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, MobiCom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[75] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Network and Distributed System Security Symposium*, NDSS '05, 2005.

[76] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, Reporting, and Fixing Performance Bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 237–246, San Francisco, CA, USA, 2013.

[77] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of*

the 2013 International Conference on Software Engineering, ICSE '13, pages 562–571, San Francisco, CA, USA, 2013.

[78] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1–6:11, Cary, NC, USA, 2012.

[79] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, Rome, Italy, 2013.

[80] Jeongyeup Paek, Joongheon Kim, and Ramesh Govindan. Energy-efficient Rate-adaptive GPS-based Positioning for Smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 299–314, 2010.

[81] Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. CarFast: Achieving Higher Statement Coverage Faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 35:1–35:11, Cary, NC, USA, 2012.

[82] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets '11, pages 5:1–5:6, Cambridge, Massachusetts, 2011.

[83] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, Bern, Switzerland, 2012.

[84] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is Keeping My Phone Awake? Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, 2012.

[85] B. Priyantha, D. Lymberopoulos, and Jie Liu. LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones. *IEEE Pervasive Computing*, 10(2):12–15, April 2011.

[86] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, Seattle, WA, USA, 2008.

[87] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Profiling Resource Usage for Mobile Applications: A Cross-layer Approach. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 321–334, Bethesda, Maryland, USA, 2011.

[88] Moo-Ryong Ra, Jeongyeup Paek, Abhishek B. Sharma, Ramesh Govindan, Martin H. Krieger, and Michael J. Neely. Energy-delay Tradeoffs in Smartphone Applications. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 255–270, San Francisco, California, USA, 2010.

[89] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 107–120, Hollywood, CA, USA, 2012.

[90] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, 2010.

[91] Kai Shen, Ming Zhong, and Chuanpeng Li. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, FAST'05, pages 23–23, San Francisco, CA, 2005.

[92] Emina Torlak and Satish Chandra. Effective Interprocedural Resource Leak Detection. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 535–544, Cape Town, South Africa, 2010.

[93] W. Visser, K. Havelund, G. Brat, and Seungjoon Park. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, ASE '00, pages 3–11, 2000.

[94] Westley Weimer and George C. Necula. Finding and Preventing Run-time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 419–431, Vancouver, BC, Canada, 2004.

[95] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 90–100, Lugano, Switzerland, 2013.

[96] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding Low-utility Data Structures. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 174–186, New York, NY, USA, 2010. ACM.

[97] Shengqian Yang, Dacong Yan, and A. Rountev. Testing for Poor Responsiveness in Android Applications. In *Proceedings of the 1st International Workshop on the Engineering of Mobile-Enabled Systems*, MOBS '13, pages 1–6, May 2013.

[98] Wei Yang, Mukul R. Prasad, and Tao Xie. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE '13, pages 250–265, Rome, Italy, 2013.

[99] S. Zaman, B. Adams, and A.E. Hassan. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 199–208, June 2012.

[100] Lide Zhang, David R. Bild, Robert P. Dick, Zhuoqing Morley Mao, and Peter Dinda. Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance. In *Proceedings of the 11th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 1–10, Sept 2013.

[101] 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 *Proceedings of the 10th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 363–372, Tampere, Finland, 2012.

[102] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, 2010.