GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications

Yepang Liu, Chang Xu, S.C. Cheung and Jian Lü

Abstract—Smartphone applications' energy efficiency is vital, but many Android applications suffer from serious energy inefficiency problems. Locating these problems is labor-intensive and automated diagnosis is highly desirable. However, a key challenge is the lack of a decidable criterion that facilitates automated judgment of such energy problems. Our work aims to address this challenge. We conducted an in-depth study of 173 open-source and 229 commercial Android applications, and observed two common causes of energy problems: missing deactivation of sensors or wake locks, and cost-ineffective use of sensory data. With these findings, we propose an automated approach to diagnosing energy problems in Android applications. Our approach explores an application's state space by systematically executing the application using Java PathFinder (JPF). It monitors sensor and wake lock operations to detect missing deactivation of sensors and wake locks. It also tracks the transformation and usage of sensory data and judges whether they are effectively utilized by the application using our state-sensitive data utilization metric. In this way, our approach can generate detailed reports with actionable information to assist developers in validating detected energy problems. We built our approach as a tool, GreenDroid, on top of JPF. Technically, we addressed the challenges of generating user interaction events and scheduling event handlers in extending JPF for analyzing Android applications. We evaluated GreenDroid using 13 real-world popular Android applications. GreenDroid completed energy efficiency diagnosis for these applications in a few minutes. It successfully located real energy problems in these applications, and additionally found new unreported energy problems that were later confirmed by developers.

Index Terms—Smartphone application, energy inefficiency, automated diagnosis, sensory data utilization, green computing.

1 Introduction

The smartphone application market is growing rapidly. Up until July 2013, the one million Android applications on Google Play store had received more than 50 billion downloads [29]. Many of these applications leverage smartphones' rich features to provide desirable user experiences. For example, Google Maps can navigate users when they hike in the countryside by location sensing. However, sensing operations are usually energy consumptive, and limited battery capacity always restricts such an application's usage. As such, energy efficiency becomes a critical concern for smartphone users.

Existing studies show that many Android applications are not energy efficient due to two major reasons [54]. First, the Android framework exposes hardware operation APIs (e.g., APIs for controlling screen brightness) to developers. Although these APIs provide flexibility, developers have to be responsible for using them cautiously because hardware misuse could easily lead to unexpectedly large energy waste [56]. Second, Android applications are mostly developed by small teams without dedicated quality assurance efforts. Their developers rarely exercise due diligence in assuring energy savings.

Locating energy problems in Android applications is difficult. After studying 66 real bug reports concerning energy problems, we found that many of these problems are intermittent and only manifest themselves at certain application states (details are given later in Section 3). Reproducing these energy problems is labor-intensive. Developers have to extensively test their applications on different devices and perform detailed energy profiling. To figure out the root causes of energy problems, they have to instrument their programs with additional code to log execution traces for diagnosis. Such a process is typically time-consuming. This may explain why some notorious energy problems have failed to be fixed in a timely fashion [15], [40], [47].

In this work, we set out to mitigate this difficulty by automating the energy problem diagnosis process. A key research challenge for automation is the lack of a decidable criterion, which allows mechanical judgment of energy inefficiency problems. As such, we started by conducting a large-scale empirical study to understand how energy problems have occurred in real-world smartphone applications. We investigated 173 open-source and 229 commercial Android applications. By examining their bug reports, commit logs, bug-fixing patches, patch reviews and release logs, we made an interesting observation: *Although the root causes of energy problems can vary with different applications, many of them (over 60%) are closely related to two types of problematic coding phenomena*:

Missing sensor or wake lock deactivation. To use a smartphone sensor, an application needs to register a listener with the Android OS. The listener should be unregistered when the concerned sensor is no longer being used. 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 [5], [8].

Sensory data underutilization. Smartphone sensors probe their environments and collect sensory data. These data are obtained at high energy cost and therefore should be utilized effectively by applications. Poor sensory data utilization can also result in energy waste. For example, Osmdroid, a popular navigation application, may continually collect GPS data simply to render an invisible map [51]. This problem occurs occasionally at certain application states. Battery energy is thus consumed, but collected GPS data fail to produce any observable user benefits.

With these findings, we propose an approach to automatically diagnosing such energy problems in Android applications. Our approach explores an Android application's state space by systematically executing the application using Java PathFinder (JPF), a widely-used model checker for Java programs [67]. It analyzes how sensory data are utilized at each explored state, as well as monitoring whether sensors/wake locks are properly used and unregistered/released. We have implemented this approach as an 18 KLOC extension to JPF. The resulting tool is named GreenDroid. As we will show in our later evaluation, GreenDroid is able to analyze the utilization of location data for the aforementioned Osmdroid application over its 120K states within three minutes, and successfully locate our discussed energy problem. To realize such efficient and effective analysis, we need to address two research issues and two major technical issues as follows.

Research issues. While existing techniques can be adapted to monitor sensor and wake lock operations to detect their missing deactivation, how to effectively identify energy problems arising from ineffective uses of sensory data is an outstanding challenge, which requires addressing two research issues. First, sensory data, once received by an application, would be transformed into various forms and used by different application components. Identifying program data that depend on these sensory data typically requires instrumentation of additional code to the original programs. Manual instrumentation is undesirable because it is labor-intensive and error-prone. Second, even if a program could be carefully instrumented, there is still no well-defined metric for judging ineffective utilization of sensory data automatically. To address these research issues, we propose to monitor an application's execution and perform dynamic data flow analysis at a bytecode instruction level. This allows sensory data usage to be continuously tracked without any need for instrumenting the concerned programs. We also propose a state-sensitive metric to enable automated analysis of sensory data utilization and identify those application states whose sensory data have been underutilized.

Technical issues. JPF was originally designed for analyzing conventional Java programs with explicit control flows [67]. It executes the bytecode of a target Java program in its virtual machine. However, Android applications are event-driven and depend greatly on user interactions. Their program code comprises many loosely coupled event handlers, among which no explicit control flow is specified. At runtime, these event handlers are called by the Android framework, which builds on hundreds of native library classes. As such, applying JPF to analyze Android applications requires: (1) generating valid user interaction events, and (2) correctly scheduling event handlers. To address the first technical issue, we propose to analyze an Android application's GUI layout configuration files, and systematically enumerate all possible user interaction event sequences with a bounded length at runtime. We show that such a bounded length does not impair the effectiveness of our analysis, but instead helps quickly explore different application states and identify energy problems. To address the second technical issue, we present an application execution model derived from Android specifications. This model captures application-generic temporal rules that specify calling relationships between event handlers. With this model, we are able to ensure an Android application to be exercised with correct control flows, rather than being randomly scheduled on its event handlers. As we will show in our later evaluation, the latter brings almost no benefit to the identification of energy problems in Android applications.

In summary, we make the following contributions in this article:

- We empirically study real energy problems from 402 Android applications. This study identifies two major types of coding phenomena that commonly cause energy problems. We make our empirical study data public for research purposes [31].
- We propose a state-based approach for diagnosing energy problems arising from sensory data underutilization in Android applications. The approach systematically explores an application's state space for such diagnosis purpose.
- We present our ideas for extending JPF to analyze general Android applications. The analysis is based on a derived application execution model, which can also support other Android application analysis tasks.
- We implement our approach as a tool, GreenDroid, and evaluate it using 13 real-world popular Android applications. GreenDroid effectively detected 12 real energy problems that had been reported, and further found two new energy problems that were later confirmed by developers. We were also invited by developers to make a patch for one of the two new problems and the patch was accepted. These evaluation results confirm GreenDroid's effectiveness and practical usefulness.

In a preliminary version of this work [42], we demonstrated the usefulness of sensory data utilization analysis in helping developers locate energy problems in Android applications. In this article, we significantly extend its earlier version in five aspects: (1) adding a comprehensive empirical study of real energy problems collected from 402 Android applications (Section 3); (2) formalizing the methodology of systematically exploring an Android application's state space for analyzing sensory data utilization (Section 4.2); (3) enhancing our sensory data utilization analysis with an outcome-based strategy, thus eliminating human effort previously required for setting algorithm parameters (Sections 4.4.3 and 6.1); (4) enhancing our evaluation with more real-world application subjects, research questions and result analyses (Section 5); (5) extending discussions of related research (Section 6).

The rest of this article is organized as follows. Section 2 introduces the basics of Android applications. Section 3 presents our empirical study of real energy problems found in Android applications. Section 4 elaborates on our energy efficiency diagnosis approach. Section 5 introduces our tool implementation and evaluates it with real application subjects. Section 6 discusses related work, and finally Section 7 concludes this article.

Table 1. Project statistics of our studied Android applications

Application type	Application availability				Application downloads			Covered categories
	Google Code	GitHub	SourceForge	Google Play	Min.	Max.	Avg.	
34 open-source applications with reported 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 applications without reported energy problems	108/139	26/139	10/139	102/139	1K ~ 5K	50M ~ 100M	0.50M ~ 1.22M	24/32
229 commercial applications with energy problems	All 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 are a total of 32 different categories of Android applications [28].

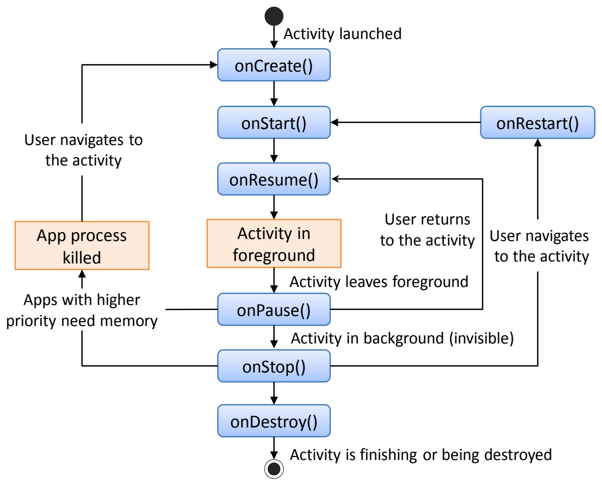

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
User returns to the activity
User navigates to the activity
App process killed
Activity in foreground
Activity leaves foreground
Apps with higher priority need memory
onPause()
Activity in background (invisible)
onStop()
onDestroy()
Activity is finishing or being destroyed

Figure 1. An activity's lifecycle diagram

2 Background

We select the Android platform for our study because it is currently one of the most widely adopted smartphone platforms and it is open for research [3]. Applications running on Android are primarily written in Java programming language. 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. For ease of presentation, we in the following may simply refer to "Android application" by "application" when there is no ambiguity. An Android application typically comprises four kinds of components as follows [3]:

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.

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 library 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 1 shows an activity's lifecycle [2]. 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.

3 Empirical Study

In this section, we report our findings from an archival study of real energy problems in Android applications. For ease of presentation, we may use "energy problems" and "energy bugs" interchangeably in subsequent discussions. 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 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 common causes of energy problems? What patterns can we distill from them to enable automated diagnosis of these problems?*

Subject selection. 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 [28] using a web crawling tool [14]. 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

Table 3. Diagnosis and fixing efforts for energy bugs in open-source Android applications

Application name	Downloads	Issue information				Diagnosis and fixing efforts			
		Issue no.	Severity level	Fixed revision no.	Inefficient revision size (LOC)	Issue open duration (Days)	# of revisions to fix	# of changed classes	# of changed methods
DroidAR[1]	5K[3] ~ 10K	27*	Medium	207	18,106	7	3	4	18
Recycle Locator	1K ~ 5K	33*	Medium	69	3,241	1	1	1	5
Sofia Public Transport Nav.	10K ~ 50K	38*	Medium	156	1,443	19	2	3	7
Sofia Public Transport Nav.	10K ~ 50K	76*	Critical	156	1,649	1	1	1	1
Google Voice Location[6]	10K ~ 50K	4*	Medium	20	4,632	330	10	4	37
BitCoin Wallet	10K ~ 50K	86	Medium	1bbc6295083c	27,220	30	1	2	4
Osmdroid	10K ~ 50K	53*	Medium	751	13,385	243	1	1	4
Osmdroid	10K ~ 50K	76*	Medium	315	8,636	11	1	1	5
Zmanim	10K ~ 50K	50/56*	Critical	323	4,807	35	1	6	14
Transdroid	10K ~ 50K	19*	Medium	Version 0.8.0	11,715	9	1	1	7
Geohash Droid	10K ~ 50K	24*	Medium	6d8f10153a48	6,682	3	1	1	6
AndTweet[6]	10K ~ 50K	29*	Medium	4a1f1f9683f2	8,908	240	1	6	22
K9Mail	1M[3] ~ 5M	574	Medium	933	72,723[5]	101	1	2	9
K9Mail	1M ~ 5M	864	Medium	317	72,723	49	3	6	8
K9Mail	1M ~ 5M	1031	Medium	1395s	72,723	20	1	1	1
K9Mail	1M ~ 5M	1643/1694	Medium	1731	72,723	6	2	3	2
K9Mail	1M ~ 5M	N/A[4]	N/A	4542e64	72,723	N/A	1	1	2
Open-GPSTracker[6]	100K ~ 500K	70	Critical	33f6e78aad9a	4,447	2	1	3	9
Open-GPSTracker[6]	100K ~ 500K	128*	Low	3aa9fb4d4ffb	9,174	9	5	7	8
Ebookdroid	500K ~ 1M	23*	Medium	138	14,351	2	1	4	5
CSipSimple	500K ~ 1M	1674	Critical	1386	54,966	6	1	1	1
c:geo[2]	1M ~ 5M	1709	Critical	cecda72	33,514	16	1	2	9
BableSink[6]	1K ~ 5K	N/A*	N/A	9fbcbf01ce	1,718	N/A	1	1	1
CWAC-Wakeful	1K ~ 5K	N/A*	N/A	c7d440f115	896	N/A	1	1	1
Ushahidi[6]	10K ~ 50K	N/A*	N/A	337b48f	10,186	N/A	1	2	9

[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 application except Google Voice Location, AndTweet, Open-GPSTracker, BabbleSink and Ushahidi are still actively maintained (continuous code revisions).

Table 2. Top five categories of inefficient commercial subjects

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

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 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 [26], GitHub [27] and SourceForge [63]. 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 (popularity), (2) have a public bug tracking system (traceability), and (3) have multiple versions (maintainability). 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).

Table 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 and four postgraduate students with a manual effort of about 35 person-weeks. We report our findings below.

3.1 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*. Be-

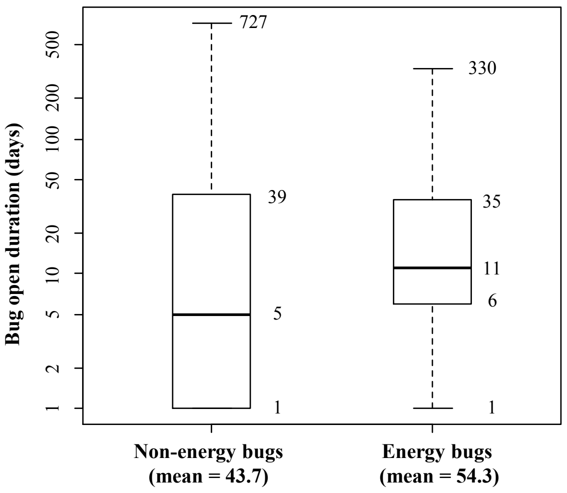


Figure 2. Open duration of energy and non-energy bugs

sides 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 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 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.*

3.2 Diagnosis and 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] We report our findings in Table 3. For each fixed energy bug, Table 3 reports: (1) bug ID, (2) severity level, (3) revision in which the bug was fixed, (4) program size of the inefficient revision, (5) duration in which the bug report is open, (6) number of revisions for fixing the bug, and (7) 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 duration of their related discussions. From these studies, we made the following three observations.

First, 24 out of the 25 energy problems listed in Table 3 are serious problems whose severity ranges from medium to critical. Developers take, on average, 54 workdays 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 workdays. Figure 2 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 [44] 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 [15], [40], [47]. 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 [15].

Third, as shown in Table 3, 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 [39]. This may become additional overhead for smartphone users when they report energy problems.

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

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 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. Developer comments on 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.*

3.3 Common Patterns of Energy Problems

Energy inefficiency is a non-functional issue whose causes can be complex and application-specific. For example, CSipSimple issue 1674 [17] happened because the application monitored too many broadcasted messages, and its issue 744 [16] was caused by unnecessary talking with a verbose server. Nevertheless, by studying the bug-fixing code 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 Table 3.

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. 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 [60].[2] We then transformed their Dalvik bytecode (contained in the .apk files) to Java bytecode using dex2jar [19], a popular Dalvik bytecode retargeting tool [49]. 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.

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 [5]. A 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 and wasted sensing operations [5]. 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 [8]. For example, Figure 3 gives a developer's comment on an energy problem in AndTweet, a Twitter client [9]. 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 4(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 4(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

[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.

[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 [9].

```
 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 each alive remote listener 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 location 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 4. Motivating examples for sensory data underutilization energy problems

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 [25]. 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 of this problem and chose to reduce the GPS sensing rate when there is no active remote listener for such location updates. Figure 3 shows their comment after fixing this energy problem.

Another interesting example is the Osmdroid problem mentioned in Section 1. Figure 4 (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 [51]. 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 prob-

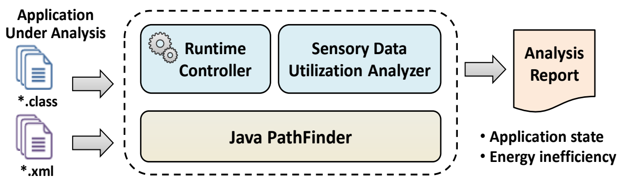


Figure 5. Approach overview

lems 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.

3.4 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.

4 Energy Efficiency Diagnosis

In this section, we elaborate on our energy efficiency diagnosis approach.

4.1 Overview

Our diagnosis is based on dynamic information flow analysis [35]. Figure 5 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 [49]. 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 in JPF's Java virtual machine, and 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? 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.

4.2 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. 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* $\overline{seq} = [e_1, e_2, \dots, e_n]$, 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 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 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):[4] 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

[4] 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 this work.

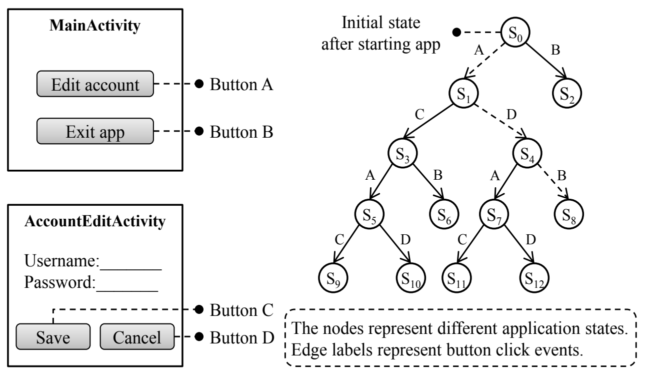


Figure 6. Illustration of event sequence generation

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:

$$S_b = \cup_{t \in T_b} S_t.$$

After defining the bounded state space exploration concept, we proceed to introduce our diagnosis approach. To effectively explore an Android application's state space, we need to generate event sequences of user interactions and schedule corresponding event handlers. These two technical issues are addressed below.

Event sequence generation. Our runtime controller, as illustrated in Figure 5, 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 6 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 6 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. 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 [56]. 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

Table 4. Example temporal rules

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

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.*

We give some examples of temporal rules in Table 4. For the entire collection of 29 rules,[5] readers may refer to our technical report [41]. 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 [23], 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 explanations for the rules in Table 4. 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 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.[6] 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.

4.3 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.) [66]. 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 [7], 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 5.2.1. Based on the preceding state exploration efforts, we can now adapt existing resource leak detection techniques [10], [68] to detect missing sensor or wake lock deactivation. In particular, our diagnosis monitors the execution of an Android application and keeps checking the violation of the following two policies:

- **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.

4.4 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.

[5] 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.

[6] 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.

Table 5. Taint propagation policy

Index	Bytecode instruction type	# instructions	Instruction semantics	Taint propagation rule
1	***Const-op*** *C*	15	$stack[0] \leftarrow C$	$T(stack[0]) = \emptyset$
2	***Load-op*** *index*	25	$stack[0] \leftarrow localVar_{index}$	$T(stack[0]) = T(localVar_{index})$
3	***LoadArray-op*** *arrayRef, index*	8	$stack[0] \leftarrow arrayRef[index]$	$T(stack[0]) = T(arrayRef) \cup T(arrayRef[index])$
4	***Store-op*** *index*	25	$localVar_{index} \leftarrow stack'[0]$	$T(localVar_{index}) = T(stack'[0])$
5	***StoreArray-op*** *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-op*** *index*	1	$stack[0] \leftarrow stack'[0].instanceField$	$T(stack[0]) = T(stack'[0].instanceField) \cup T(stack'[0])$
9	***GetStatic-op*** *index*	1	$stack[0] \leftarrow ClassName.staticField$	$T(stack[0]) = T(ClassName.staticField)$
10	***PutField-op*** *index*	1	$stack'[1].instanceField \leftarrow stack'[0]$	$T(stack'[1].instanceField) = T(stack'[0])$
11	***PutStatic-op*** *index*	1	$ClassName.staticField \leftarrow stack'[0]$	$T(ClassName.staticField) = T(stack'[0])$
12*	***Return-op(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*)**
<u>1</u>	Push a constant value *C* onto the operand stack (*stack*[0] represents the value at the stack top after an operation).
<u>2</u>, 3	Load the value of the #*index* local variable onto the operand stack.
<u>4</u>, 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).
<u>6</u>, 7	Perform the binary operation ⊗ on the two values popped from the operand stack (i.e., *stack'*[0] and *stack'*[1]), and push the result back onto stack.
<u>8</u>, 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.
<u>10</u>, 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.
<u>12</u>	Pop the value at the callee's operand stack top (i.e., *calleeStack'*[0]), and push the value onto the caller's operand stack.

Notes: (1) 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 [22]. 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 7 for illustration). (2) Rule 12 does not conflict with the rule for handling control dependencies (see Section 4.4.2). They can be applied together.

4.4.1 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.

4.4.2 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 4(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.[7] 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 5 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 5 is designed for a set of bytecode instructions with similar semantics (explained in the lower part of Table 5). 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"

[7] 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.

```
 1.  public void onSensorChanged(SensorEvent event){
 2.    if(event.sensor.getType() == Sensor.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 = (x * x + y * y + z * z) / (g * g);
27.    updateAccTextView(accelerationSquareRoot);
28.    if(accelerationSquareRoot >= 2){
29.        return true;
30.    }
31.    return false;
32.  }
```

Figure 7. Example code to demonstrate taint propagation

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 (at the top of the operand stack after the calculation, represented by *stack*[0] in Table 5) 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 5) is tainted before calculation. Other taint propagation rules can be explained similarly.

We illustrate the taint propagation process by a concrete example. Figure 7 lists the code snippet from an application that uses accelerometer data to compute and display a phone's current acceleration status (Lines 21–27). 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 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). 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 [12], [62]. 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 [22], [37]. Our taint propagation terminates when the application under analysis finishes its handling of sensor event.[8] 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 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.

4.4.3 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,[9] 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

[8] 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.

[9] 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 a message broadcast API.

Table 6. GPS data utilization coefficients at three states

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.

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 (1):

$$DUC(s, d) = \frac{usage(s, d)}{Max_{s' \in S_b, d' \in D}(usage(s', d'))} \quad (1)$$

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 the following equation:

$$usage(s, d) = \sum_{i \in API\_Call(s,d)} eTest(i, d, s) \times noInst(i) \quad (2)$$

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 observable outcomes/benefits to users (e.g., updating visible GUIs or writing to file systems). Specifically, our 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.
- 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.

==
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 8. Example energy problem report

As such, our analysis can identify those application states where sensory data are underutilized based on calculated sensory data usage and cross-state comparison. We give one example for illustration. Consider the three states in the Osmdroid example in Figure 4(b). They are also listed in Table 6. 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 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 (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 6. 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 8. It shows that GPS data are not well-utilized by Osmdroid at the second application state described in Table 6. 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 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 [54]. 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.

5 Experimental Evaluation

We implemented our energy diagnosis approach as a prototype tool named GreenDroid on top of JPF [31]. 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 [45]. These library classes are mostly built on native code. Due to JPF's closed-world assumption [67], 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 [24]. 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 [31], [41]. 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) [53], 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 registered broadcast receivers, and (4) a list of registered sensor listeners and wake locks. More tool implementation details can be found in our technical report [41].

In this section, we evaluate GreenDroid by controlled experiments. We aim to answer the following four 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 for generated user interaction event sequences affect the thoroughness of our energy diagnosis in terms of code coverage?*
- **RQ7 (Comparison with existing resource leak detection work):** *How does GreenDroid compare to existing resource leak detection work in terms of finding real missing sensor or wake lock deactivation problems?*

Table 7. Experimental subject information and detected energy problem

Application Name	Version	Lines of code	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)
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.

5.1 Experimental Setup

We selected 13 open-source Android applications as our experimental subjects. Table 7 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. The first 11 applications were confirmed to have energy problems of our two identified patterns (Section 3.3). We then 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 development descriptions, 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 7 that our selected applications have been popularly downloaded (over 1 million downloads in total), and covered a variety of application categories (10 different categories). We obtained these applications' source code and compiled them on Android 2.3.3 for our experiments. We chose Android 2.3.3 because it is one of the most widely adopted Android platforms and is compatible with most applications on the market [6]. 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 four research questions in turn.

5.2 RQ4: Effectiveness and Efficiency

To answer research question RQ4 about GreenDroid's effectiveness and efficiency, we ran GreenDroid to diagnose each application listed in Table 7 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 ineffective API calls, to see whether they can locate real energy problems in these applications.

We observed that GreenDroid successfully located 14 real energy problems in these applications, as listed in Table 7. Four of them are caused by missing sensor deactivation, four by missing wake lock deactivation, and the remaining six by sensory data underutilization. As mentioned earlier, the first 12 energy problems listed in Table 7 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 7, the severity levels of our detected 14 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.

5.2.1 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 [5], [8]. However, we found that missing sensor or wake lock deactivation is common in Android applications. GreenDroid detected eight applica-

```
/**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 9. The energy problem in Ushahidi application

tions suffering such energy problems from our 13 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 9 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 previous registered *gpsListener* to the sensor listener unregistration API removeUpdate(), 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 [7]. 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 9 also gives the correct version for comparison.

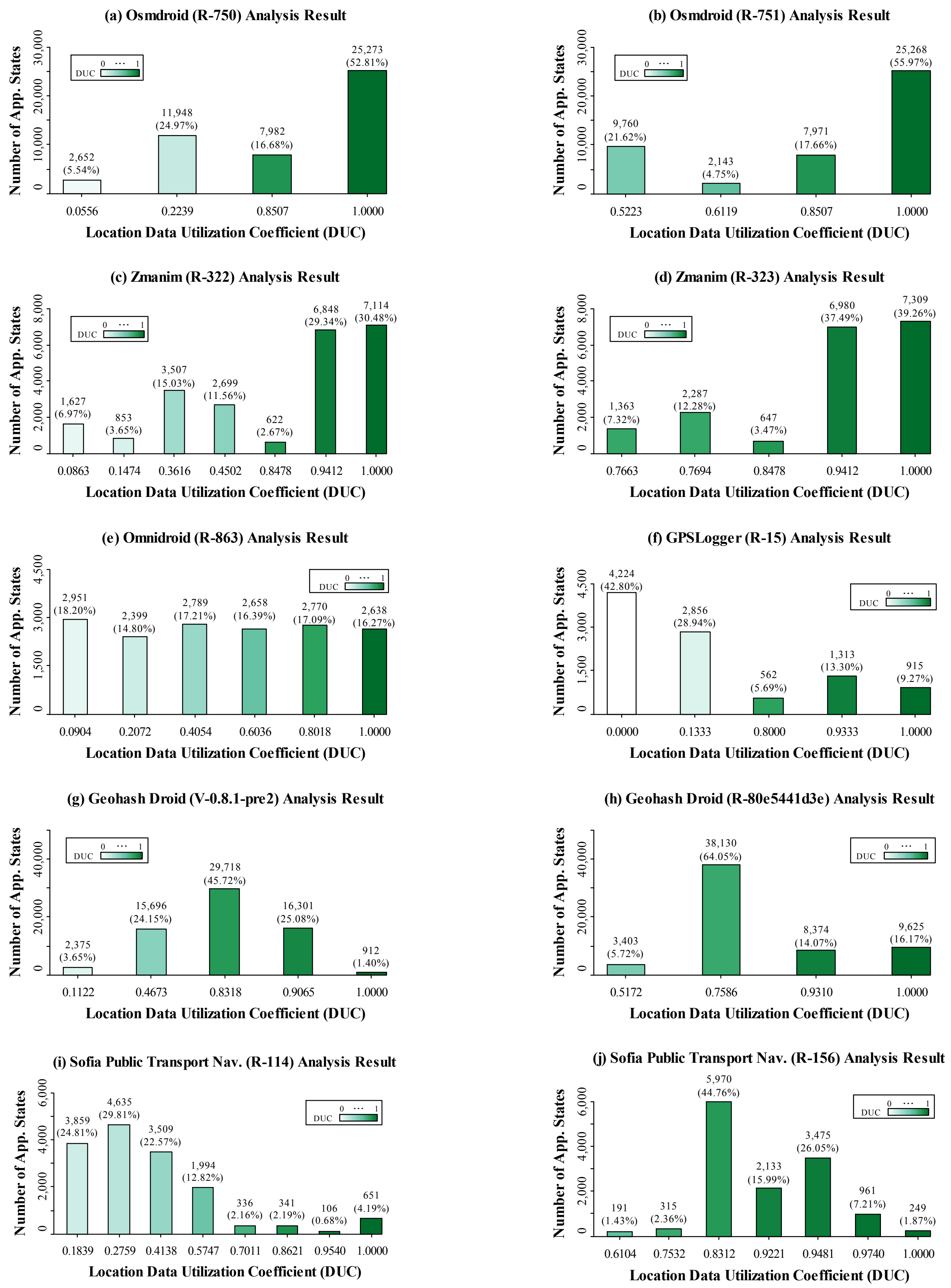
5.2.2 Sensory data underutilization

GreenDroid also detected six applications suffering sensory data underutilization problems out of our 13 subjects. Among these detected problems, three (Table 7) are critical ones that can cause massive energy waste. We discuss these six 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 10(a). 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 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 [51]. To fix this problem, developers later disabled GPS sensing if users leave MapActivity without the location tracking functionality enabled. Figure 10(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 (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 10(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 [72]. 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 10(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."*



Notes: (1) In the above 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 10. Sensory data utilization analysis results for six applications

Omnidroid. Omnidroid helps automate system functionalities based on user contexts. For example, Omnidroid can help users automatically send a reply message 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 prespecified condition, its corresponding action would be executed as a response. Our diagnosis results in Figure 10(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" [50]:

> **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 10(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 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 [30]. We will further discuss our patch later in Section 5.6.

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 10(g). We studied 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). 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 [25]. Figure 3 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 [25]. 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 10(h) 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 10(i). We examined 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,[10] 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 [64] 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 [65]. 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 10(j) shows, there are now no application states

[10] 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 8. 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
Omnidroid	52,805	78	242	396	22/100
GPSLogger	58,824	28	41	153	9/100

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 diagnosis 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.

5.2.3 Analysis overhead

Table 8 presents GreenDroid's diagnosis overhead. For each of our 13 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,[11] (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 8 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 Omnidroid (over 12K LOC) and DroidAR (over 18K LOC), GreenDroid explored over 50K states during its diagnosis and executed over 60 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 400 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 [61]. This suggests that GreenDroid is a practical tool for diagnosing energy problems in real-world Android applications.

5.3 RQ5: 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 average 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 [4], 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

[11] 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.

Table 9. Statement coverage with respect to different event sequence length limit settings

Application names	# activity components	Statement coverage (%)								
		Limit = 0	Limit = 1	Limit = 2	Limit = 3	Limit = 4	Limit = 5	Limit = 6	Limit = 7	Limit = 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
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.

them with those from GreenDroid. We discuss these experimental results 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 8 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 exception 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 remaining 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.

5.4 RQ6: 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 [11], [52]. Table 9 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 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 9, 8 of our 13 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 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 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 and Omnidroid, GreenDroid covers less than 25% statements. We thus inspected these three applications and found three major difficulties in achieving higher code coverage. These findings can benefit related research such as automated Android application testing [11], [34]. 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 [1], 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 [45]. 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 (Sections 1 and 3), 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 candidate for assessing the effectiveness of such energy diagnosis.

5.5 RQ7: Comparison with Existing Resource Leak Detection Work

Our work shares some similarity with existing resource leak detection work [10], [32], [66], [68] since sensor listeners and wake locks are considered as valuable resources in Android OS and applications. Our last 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 [32]. Relda is the latest resource leak detection work dedicated for Android applications [32]. 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). We applied Relda to analyze all 13 application subjects listed in Table 7. It 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 this article. We further invited Relda's authors to manually validate these raw data and remove duplicate and false warnings as they did in their publication [32] (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 our 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 this article, 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 3.3 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 point-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 point-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 9), instead of passing the listener that has been registered earlier (Line 6 in Figure 9). (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, but Google Maps APIs have wrapped GPS sensor listener registration/unregistration APIs such that the latter cannot be examined by Relda. 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 point-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 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 article. 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 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 [66]). 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.

5.6 Discussions

Patching GPSLogger. As mentioned earlier, we were invited by GPSLogger developers to make a patch to improve GPSLogger's energy efficiency. To be realistic, we built this patch by following an accepted patch for fixing Geohash Droid's energy problem [25]. Our patch slightly modifies GPSLogger's GPS sensing part, aiming not to affect its functionalities. Specifically, the patched GPSLogger would slow down its GPS sensing rate to every 30 seconds when it finds that its collected GPS data remain at low quality (e.g., after five consecutive imprecise readings), and set the sensing rate immediately back to the original value when it finds that its GPS data have become precise again (e.g., after two consecutive precise readings). We submitted this patch to GPSLogger developers and it was recently accepted. We also helped release it online for trial downloads for interested users.[12] So far, this patch has received more than 400 downloads. This indicates that developers indeed acknowledge and accept our efforts in helping defend their Android applications from energy inefficiency.

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 article. 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 [45]. 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. 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 [22]. 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 [62]. 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) [37]. It is therefore suggested that the tainting policy should be designed according to its application scenarios [62]. 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 [22] 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. Second, GreenDroid's event sequence generation belongs to the category of model-based approaches [34], [45], [69]. 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. Third, GreenDroid can-

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

not 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 [61] 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 [34]). 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 (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 4(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.

6 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.

6.1 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 [18] helped offload "energy-consuming" tasks to resource-rich infrastructures such as remote servers. EnTracked [38] and RAPS [57] adopted different heuristics to guide an application to use GPS sensors in a smart way. Little Rock [58] suggested a dedicated low power processor for energy-consuming sensing operations. SALSA [59] 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 [36]. 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) [54], [56]. Zhang et al. proposed a taint-tracking technique for the Android platform to detect energy wastes caused by unnecessary network communications [70]. 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) [43]. 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 [43], [56], [70]. Nevertheless, our work differs from them on several aspects. Regarding Pathak et al's work [56], 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 this article) 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 [70]. 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 [43]. 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.

Our preliminary version of this work (i.e., our earlier conference paper [42]) has shown that sensory data utilization analysis can help locate energy problems caused by cost-ineffective use of sensory data. In this article, we enhanced our sensory data utilization analysis algorithm by addressing two issues in our earlier analysis discussed in the conference paper. First, our earlier analysis considers intermediate computational instructions as legitimate utilization of sensory data, but these instructions' execution may not produce perceptible benefits for users. For example, consider the following two scenarios, which could occur in reality (although we did not see such examples in our experiments). In the first scenario, an application receives raw GPS data. It conducts non-trivial intermediate computation to process these data, but the processed data are not used afterwards. In the second scenario, GPS data are slightly processed before they are utilized to render visible GUI elements for user interaction. From this example, we can see that the battery power is clearly wasted in the first scenario. However, our earlier analysis would consider that sensory data have been more effectively utilized in the first scenario than the second due to the non-trivial intermediate computation involved. Another issue is that our earlier analysis requires assigning a weighting function for each instruction that uses sensory data. The determination of such weighting functions may not be obvious for Android developers and can vary across different applications. Therefore, to address these two issues, we enhanced our analysis algorithm in this article by considering only those instructions that consume sensory data and produce observable benefits to users as legitimate utilization of sensory data. The new analysis algorithm can successfully identify the first scenario in the above example as a problematic scenario. Besides, since the algorithm makes a binary decision when judging whether sensory data are effectively used by an application, our analysis no longer depends on weighting functions whose weighting factors may require manual customization effort.

6.2 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 [71] 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 [21] shares the same goal as PowerTutor, but can perform energy estimation for much smaller time intervals (e.g., as small as 10ms). eProf [55] 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 [46] 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 [33]. 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 [25].

6.3 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 [68]. To prevent resource leaks, researchers proposed language-level mechanisms and automated management techniques [20]. Various tools were also developed to detect resource leaks [10], [66]. For example, QVM [10] is a specialized runtime environment for detecting defects in Java programs. It monitors application executions and checks for violations of resource safety policies. TRACKER [66] 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 [32]. Similar to our work, they also adapted the general idea of resource safety policy checking discussed in QVM [10] and TRACKER [66] 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 article, 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.

6.4 Information Flow Tracking

Dynamic information flow tracking (DFT for short) observes interesting data as they propagate in a program execution [35]. DFT has many useful applications. For example, TaintCheck [48] 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 [22] 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 [13] 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.

7 CONCLUDING REMARKS

In this article, 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 13 real-world popular Android applications. Our experimental results confirmed the effectiveness and practical usefulness of GreenDroid.

In future, we plan to study more Android applications and identify other common causes of energy problems. For example, we found from our study that a non-negligible proportion (about 16%) of energy problems was caused by network issues (e.g., energy-inefficient data transmission). We are going to study these issues to further extend our approach. By doing so, we expect that our research will help advance energy efficiency practices for a wider range of smartphone applications, and thus potentially benefit millions of smartphone users.

ACKNOWLEDGEMENT

This research was partially funded by Research Grants Council (General Research Fund 611813) of Hong Kong, National High-Tech Research & Development Program (863 program 2012AA011205), and National Natural Science Foundation (61100038, 91318301, 61361120097) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486). Preliminary ideas of this work have been discussed in a conference paper (PerCom 2013) [42]. We greatly thank anonymous reviewers for their valuable comments for improving the conference version and this extended version. Besides, we would also like to sincerely thank Relda's authors [32] for their generous help during our experiments.

References

[1] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Soft. Engr. (FSE 12), ACM, 2012, pp. 59:1-59:11.

[2] "Android Activity Lifecycles." URL: http://developer.android.com/guide/components/activities.html.

[3] "Android Developers Webstie." URL: http://developer.android.com/index.html.

[4] "Android Emulator." URL: http://developer.android.com/tools/help/emulator.html.

[5] "Android Sensor Management." URL: http://developer.android.com/reference/android/hardware/SensorManager.html.

[6] "Android Platform Versions." URL: http://developer.android.com/about/dashboards/index.html.

[7] "Android Process Lifecycle." URL: http://developer.android.com/reference/android/app/Activity.html#ProcessLifecycle.

[8] "Android Power Management." URL: http://developer.android.com/reference/android/os/PowerManager.html.

[9] "AndTweet issue 29." URL: https://code.google.com/p/andtweet/issues/detail?id=29.

[10] M. Arnold, M. Vechev, and E. Yahav, "QVM: an efficient runtime for detecting defects in deployed systems," ACM Trans. Software Engineering and Methodology, vol. 21, 2011, pp. 2:1-2:35.

[11] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," Proc. ACM SIGPLAN Int'l Conf. Object-oriented Programming Systems Languages & Applications (OOPSLA 13), ACM, pp. 641-660.

[12] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," Proc. Int'l Symp. Software Testing and Analysis (ISSTA 07), 2007, pp. 196-206.

[13] J. Clause and A. Orso, "LEAKPOINT: pinpointing the causes of memory leaks," Proc. Int'l Conf. Soft. Engr. (ICSE 10), pp. 515-524.

[14] "Crawler4j." URL: https://code.google.com/p/crawler4j/.

[15] "CSipSimple issue 81." URL: https://code.google.com/p/csipsimple/issues/detail?id=81.

[16] "CSipSimple issue 744." URL: https://code.google.com/p/csipsimple/issues/detail?id=744.

[17] "CSipSimple issue 1674." URL: https://code.google.com/p/csipsimple/issues/detail?id=1674.

[18] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 10), ACM, 2010, pp. 49-62.

[19] "dex2jar." URL: https://code.google.com/p/dex2jar/.

[20] I. Dillig, T. Dillig, E. Yahav, and S. Chandra, "The CLOSER: automating resource management in Java," Proc. Int'l Symp. Memory Management (ISMM 08), ACM, 2008, pp. 1-10.

[21] M. Dong and L. Zhong, "Sesame: Self-constructive high-rate system energy modeling for battery-powered mobile systems," Proc. Int'l Conf. Mobile Systems, Applications, and Services (Mobisys 11), ACM, 2011, pp. 335-348.

[22] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," Proc. USENIX Conf. Operating Systems Design and Impl. (OSDI 10), 2010, pp. 393-407.

[23] K. Etessami, and T. Wilke, "An Until hierarchy for temporal logic," Proc. IEEE Symp. Logic in Computer Science (LICS 96). 1996, pp. 108-117.

[24] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," Proc. ACM Conf. Computer and Communications Security (CCS 11), 2011, pp. 627-638.

[25] "GeoHashDroid issue 24." URL: https://code.google.com/p/geohashdroid/issues/detail?id=24.

[26] "Google Code website." URL: http://code.google.com/.

[27] "GitHub website." URL: https://github.com/.

[28] "Google Play store website." URL: https://play.google.com/store.

[29] "Google Play Wiki Page." URL: http://en.wikipedia.org/wiki/Google_Play.

[30] "GPSLogger issue 7." URL: https://code.google.com/p/gpslogger/issues/detail?id=7.

[31] "GreenDroid project." URL: http://sccpu2.cse.ust.hk/greendroid (id: greendroid, password: tsesubmission).

[32] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in Android applications," Proc. ACM/IEEE Int'l Conf. Automated Soft. Engr. (ASE 13), 2013, pp. 389-398.

[33] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," Proc. 35th Int'l Conf. Soft. Engr. (ICSE 13), pp. 92-101.

[34] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," Proc. Int'l Symp. Software Testing and Analysis (ISSTA 13), 2013, pp. 67-77.

[35] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," Proc. ACM Conf. Virtual Exe. Env. (VEE 12), 2012, pp. 121-132.

[36] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," Proc. Int'l Conf. Mobile Sys., App's, and Services (MobiSys 08), 2008, pp. 239-252.

[37] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit Flows: Can't Live with `Em, Can't Live without `Em," Proc. Int'l Conf. Information Systems Security (ICISS 08), 2008, pp. 56-70.

[38] M. B. Kjærgaard, J. Langdal, T. Godsk, and T. Toftkjær, "EnTracked: energy-efficient robust position tracking for mobile devices," Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 09), ACM, 2009, pp. 221-234.

[39] "K9Mail issue 1986." URL: https://code.google.com/p/k9mail/issues/detail?id=1986.

[40] "K9Mail issue 3348." URL: https://code.google.com/p/k9mail/issues/detail?id=3348.

[41] Y. Liu, C. Xu and S.C. Cheung, "Verifying Android applications Using Java Pathfinder," Technical Report HKUST-CS-12-03.

[42] Y. Liu, C. Xu and S.C. Cheung, "Where has my battery gone? Finding sensor related energy black holes in smartphone applications," Proc. 11th IEEE Int'l Conf. Pervasive Computing and Communications (PERCOM 13), 2013, pp. 2-10.

[43] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones," Proc. 10th ACM/USENIX Symp. Networked Systems Design and Implementation (NSDI 13), April 2013, pp. 57-70.

[44] 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, vol. 18, 1947, pp. 50-60.

[45] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," SIGSOFT Softw.

Eng. Notes, vol. 37, 2012, pp. 1-5.

[46] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," Proc. 18th Int'l Conf. Mobile Computing and Networking (Mobicom 12), 2012, pp. 317-328.

[47] "MyTracks issue 520." URL: https://code.google.com/p/mytracks/issues/detail?id=520.

[48] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Proc. ISOC Network and Distributed System Security Symp., 2005.

[49] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," Proc. ACM SIGSOFT Int'1 Symp. Foundations of Soft. Engr. (FSE 12), ACM, 2012, pp. 6:1-6:11.

[50] "Omnidroid issue 179." URL: https://code.google.com/p/omnidroid/issues/detail?id=179.

[51] "Osmdroid issue 53." Available: http://code.google.com/.

[52] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "CarFast: achieving higher statement coverage faster," Proc. ACM SIGSOFT Int'l Symp. Foundations of Soft. Engr. (FSE 12), ACM, 2012, pp. 35:1-35:11.

[53] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," Proc. Int'l Symp. Software Testing and Analysis (ISSTA 08), ACM, 2008, pp. 15-26.

[54] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," Proc. ACM Workshop on Hot Topics in Networks (Hotnets 11), ACM, 2011, pp. 5:1-5:6.

[55] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof," Proc. Euro. Conf. Comp. Sys. (EuroSys 12). 2012, pp. 29-42.

[56] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps," Proc. 10th Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 12), 2012, pp. 267-280.

[57] J. Paek, J. Kim, and R. Govindan, "Energy-efficient rate-adaptive GPS-based positioning for smartphones," Proc. Int'l Conf. Mobile Systems, App., and Services (MobiSys 10), ACM, 2010, pp. 299-314.

[58] B. Priyantha, D. Lymberopoulos, and J. Liu, "LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones," IEEE Pervasive Computing, vol. 10, 2011, pp. 12-15.

[59] M. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely, "Energy-delay tradeoffs in smartphone applications," Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 10), ACM, 2010, pp. 255-270.

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

[61] "Robotium, a testing framework for Android applications." URL: http://code.google.com/p/robotium/.

[62] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution," Proc. IEEE Symp. Security and Privacy, 2010, pp. 317-331.

[63] "SourceForge website." URL: http://sourceforge.net/.

[64] "Sofia Public Transport Nav issue 38." URL: https://code.google.com/p/sofia-public-transport-navigator/issues/detail?id=38.

[65] "Sofia Public Transport Nav issue 76." URL: https://code.google.com/p/sofia-public-transport-navigator/issues/detail?id=76.

[66] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," Proc. Int'l Conf. Soft. Engr. (ICSE 10), 2010, pp. 535-544.

[67] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," Proc. Int'l Conf. Automated Soft. Engr., 2000, pp. 3-11.

[68] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," Proc. ACM SIGPLAN Conf. Object-oriented Prog., Sys, Lang., and App's (OOPSLA 04), 2004, pp. 419-431.

[69] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," Lecture Notes in Computer Science, vol. 7793, 2013, pp. 250-265.

[70] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang, "ADEL: an automatic detector of energy leaks for smartphone applications," Proc. 8th IEEE/ACM/IFIP Int'l Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS 12), 2012, pp. 363-372.

[71] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS 10), 2010, pp. 105-114.

[72] "Zmanim issue 50/56." URL: https://code.google.com/p/android-zmanim/issues/detail?id=50/56.