NavyDroid: Detecting Energy Inefficiency Problems for Smartphone Applications

Yi Liu, Jue Wang, Chang Xu, and Xiaoxing Ma
State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
nettee.liu@gmail.com,juewang591@gmail.com,changxu@nju.edu.cn,xxm@nju.edu.cn

ABSTRACT

Many smartphone applications suffer from energy inefficiency problems, but locating these problems is quite difficult and labor-intensive. Automated tools for detecting energy inefficiency bugs have been shown to be effective. Existing approaches generally consist of two parts, namely the simulation part and the monitor part. The simulation part explores an application's state space guided by an application execution model, and the monitor part checks for occurrences of energy inefficiency patterns. However, existing approaches might miss energy inefficiency bugs due to their imprecise application execution models and oversimplified energy inefficiency diagnosis policies. In this paper, we proposed NavyDroid, an approach to diagnosing energy inefficiency problems more effectively. We summarized a comprehensive application execution model from Android specifications and expressed it as a state machine. By considering multiple patterns of wake lock misuses, our approach is able to detect more complex energy bugs caused by wake lock misuses. We implemented NavyDroid on top of Java Pathfinder (JPF) and applied it to real-world applications. We evaluated NavyDroid with 17 real-world Android applications, and NavyDroid located more energy inefficiency bugs in these applications than the existing work E-GreenDroid did. The results of our experiments demonstrate that our approach can effectively locate real energy inefficiency bugs in Android applications, suggesting its effectiveness.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software testing and debugging**;

KEYWORDS

energy inefficiency, smartphone application, wake lock

1 INTRODUCTION

With the rapid development of the smartphone application market, Android applications grow in number quickly. Meantime, many of these applications employ energy-consuming operations, such as location sensing, to provide a good user experience. However, if not used properly, these operations can cause unnecessary energy waste and exhaust battery power quickly. Energy inefficiency problems have become more and more common in Android applications, raising concern from users.

Locating energy inefficiency problems in Android applications is rather difficult for developers. An energy inefficiency problem often occurs at certain application states. Developers have to exhaust all possible application states to reproduce the problem and figure out the root cause. Therefore, an automatic detecting tool is in demand.

Existing approaches are devoted to detecting energy inefficiency problems automatically, and are shown to be very effective [17, 19, 25]. GreenDroid [19] detects the missing deactivation of sensors and wake locks, and analyzes the underutilization of sensor data. CyanDroid [17] proposes a novel approach to systematically generate multidimensional sensor data. E-GreenDroid [25] updates and optimizes the simulation execution and library modeling of GreenDroid, and supports some new Android features.

Generally, energy inefficiency detecting tools consist of two parts, namely the *simulation* part and the *monitor* part.

(1) **Simulation.** The *application execution engine* simulates the execution of an Android application, and explores its application states. The simulation process mainly consists of event sequence generation and state space exploration, and is guided by an *application execution model.*
(2) **Monitor.** During the simulation execution, the diagnosis system monitors energy-consuming operations, and checks whether these operations match *energy inefficiency patterns.* An investigation shows that many energy inefficiency problems associate with two types of *energy inefficiency patterns* [19]:
   - **Missing sensor listener or wake lock deactivation.** An application should register a sensor listener before receiving data from a sensor. Similarly, an application should acquire a wake lock to keep the CPU awake to complete some long-time work. The smartphone will not disable sensors or go to sleep before the sensor listener or the wake lock is deactivated. Forgetting to unregister sensor listeners or to release wake locks can consume battery power quickly [4, 9].
   - **Sensor data underutilization.** Sensor data is obtained at the expense of energy. Thus, sensor data should be utilized effectively to bring user benifits. Poor utilization of sensor data is equivalent to energy waste.

Existing approaches follow this two-part architecture. However, they have several limitations and suffer from false negatives. They can be improved in both two parts.

Internetware'17, September 23, 2017, Shanghai, China
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5313-7/17/09…$15.00
https://doi.org/10.1145/3131704.3131705

First, for simulation part, existing approaches use imprecise application execution models. The runtime behaviors of Android applications are complicated, and have some corner cases. Existing application execution models are not consistent with the actual execution of Android applications to some extent. Specifically, existing approaches fail to accurately simulate the paused state and the killed state of activities.

Second, for monitor part, existing approaches neglect some patterns of wake lock misuses. Wake locks have multiple misuse patterns, and the use of wake locks can be more complicated than that of sensor listeners. Existing approaches are unable to detect complex patterns of wake lock misuses, e.g. multiple lock acquisition.

Therefore, we propose an approach called NavyDroid to address the above issues in this paper. We construct the application execution model as a strengthened DFA, i.e. deterministic finite automaton, which can accurately simulate the paused state, the killed state, and related state transitions of an activity. We extend the misuse checker to monitor multiple wake lock misuse patterns. In addition, we optimize the generation of user events so that the simulated application can terminate normally in each execution.

We implemented NavyDroid as a prototype tool to evaluate its ability for detecting energy inefficiency problems. We selected 17 real-world Android appliations and analyzed these applications with NavyDroid and E-GreenDroid [25], a state-of-the-art energy inefficiency diagnosis tool. As a result, NavyDroid not only located all the energy inefficiency bugs that E-GreenDroid located, but also detected more energy inefficiency bugs than E-GreenDroid did. The evaluation results demonstrate that our approach can detect energy inefficiency problems more effectively.

In summary, our work makes the following contributions:

- We summerize a precise execution model for Android applications. The application execution model is more consistent with the actual execution than existing models.
- We extend NavyDroid to detect more patterns of wake lock misuses. NavyDroid is able to recognize the isuse pattern of multiple lock acquisitions and check for energy inefficiency problems related to this pattern.
- We implement NavyDroid as a prototype tool and evaluate it with real-world Android applications. NavyDroid located real energy inefficiency bugs in these applications, indicating its effectiveness.

The rest of this paper is organized as follows. Section 2 introduces the basics of Android applictions and presents a motivating example. Section 3 elaborates on our energy inefficiency diagnosis approach. Section 4 evaluates NavyDroid with real-world applications and discusses the experimental results. Section 5 reviews some related work, and Section 6 concludes this paper.

2 BACKGROUND AND MOTIVATION

In this section, we introduce the basics of Android applications, and present a motivating example of our work.

2.1 Background

Android applications are composed of different *components*. There are four different types of application components [2]:

Activity. An activity is the entry point of an application for interacting with the user. It represents a single screen with a graphical user interface (GUI). The GUI layouts corresponding to an activity are specified in configuration files. An application usually has a series of independent and collaborative activities.

Service. A service runs in the background to perform long-running operations. An activity can interact with a service when it *binds* to the service.

Broadcast receiver. A broadcast receiver is a component that responds to system-wide broadcast messages. A broadcast receiver can be another entry into the application besides activities. It often works as a gateway to other components.

Content provider. A content provider manages a shared set of application data. Through the content provider, other components and applications can query or modify the data.

Each application component follows a prescribed lifecycle when it transits through different states [1]. As a component enters a new state, the Android framework invokes its corresponding *lifecycle callbacks*. Lifecycle callbacks are a subset of *event handlers*, whose scheduling is a key part of Android applications' simulation execution.

2.2 Motivating Example

We present TomaHawk [11] as the motivating example, which has two energy inefficiency problems. Figure 1 shows a simplified version of the problematic code snippet. Two components, namely `PlaybackActivity` and `PlaybackService`, are responsible for playing the media specified by users. Users can click the play/pause button to switch between the *playing* and the *paused* mode of media playing (Lines 12–15, 48–51). The `PlaybackActivity` starts a `PlaybackService` in the background (Lines 17–19), and binds to the service for interaction (Lines 22–24). In order to prevent the media playing from being interrupted, `PlaybackService` uses a wake lock to keep the CPU on during the media playing (Lines 34–35). When the media playing starts, the wake lock will be acquired (Lines 54–56); when the media playing pauses or stops, the wake lock will be released (Lines 59–61, 64–66).

There are several wake lock operations in `PlaybackService`. When the media player gets prepared, the wake lock will also be acquired (Lines 43–46). The service checks if the wake lock is held before releasing it (Lines 61, 66), but acquires the wake lock directly without checking (Line 55), so the wake lock can be acquired more than once. If a user clicks the play/pause button twice after the media player gets prepared, the wake lock will be acquired twice but released only once. As wake locks are reference-counted by default, each call to `acquire()` must be balanced by an equal number of calls to `release()`, otherwise the wake lock will not be released [10]. In this case, the wake lock will not be released, and the battery power will continue to be consumed without user benefits. Existing tools, such as GreenDroid and E-GreenDroid, fail to detect this energy inefficiency problem, because of the lack of precise analysis on wake lock usage. They cannot analyze the misuse pattern that a wake lock is acquired more than once.

Another energy inefficiency problem occurs when the activity and the service is killed by the Android system. `PlaybackService` stops the playing of media and releases the wake lock in the `onDestroy()`

```
 1 public class PlaybackActivity extends Activity {
 2     private ImageButton mPlayPauseButton;
 3     private PlaybackService mPlaybackService;
 4     private ServiceConnection mConnection = new ServiceConnection() {
 5         public void onServiceConnected(
 6                 ComponentName name, IBinder binder) {
 7             mPlaybackService = ((MyBinder) binder).getService();
 8         }
 9     };
10     public void onCreate(Bundle state) {
11         mPlayPauseButton.setOnClickListener(new OnClickListener() {
12             public void onClick(View v) {
13                 // toggle play/pause state
14                 mPlaybackService.playPause();
15             }
16         });
17         Intent i = new Intent(this, PlaybackService.class);
18         // start PlaybackService
19         startService(i);
20     }
21     public void onResume() {
22         Intent i = new Intent(this, PlaybackService.class);
23         // bind PlaybackService
24         bindService(i, mConnection, Context.BIND_ABOVE_CLIENT);
25     }
26 }
27
28 public class PlaybackService extends Service
29         implements MediaPlayer.OnPreparedListener {
30     private static String TAG = PlaybackService.class.getName();
31     private WakeLock mWakeLock;
32     private MediaPlayer mMediaPlayer;
33     public void onCreate() {
34         mWakeLock = getPowerManager().newWakeLock(
35                 PowerManager.PARTIAL_WAKE_LOCK, TAG);
36         mMediaPlayer = createAndSetupMediaPlayer();
37         mMediaPlayer.setOnPreparedListener(this);
38     }
39     public void onDestroy() {
40         // stop media when the service is destroyed
41         stop();
42     }
43     public void onPrepared() {
44        // start media when the media player get prepared
45         start();
46     }
47     public void playPause() {
48         if (mMediaPlayer.isPlaying())
49             pause();
50         else
51             start();
52     }
53     public void start() {
54         // play media and acquire wake lock
55         mWakeLock.acquire();
56         mMediaPlayer.start();
57     }
58     public void stop() {
59         // stop media and release wake lock
60         mMediaPlayer.stop();
61         if (mWakeLock.isHeld()) mWakeLock.release();
62     }
63     public void pause() {
64         // pause media and release wake lock
65         mMediaPlayer.pause();
66         if (mWakeLock.isHeld()) mWakeLock.release();
67     }
68     public class MyBinder extends Binder {
69         PlaybackService getService() {
70             return PlaybackService.this;
71         }
72     }
73     public IBinder onBind(Intent intent) {
74         return new MyBinder();
75     }
76 }
```

Figure 1: Motivating example from the TomaHawk application (revision 543c3b9ab4)

callback (Lines 40–41). When a user presses the *back* button and exits the activity, the service will also be destroyed and the wake lock will be released. However, the Android system can kill processes when the memory is insufficient, in which case the `onDestroy()` callback of killed activities and services will not be invoked [1]. Thus, the wake lock will not be released, affecting the battery life. GreenDroid and E-GreenDroid, with imprecise application execution models, cannot simulate the runtime behaviors of activities or services being killed, so they fail to detect this energy inefficiency problem.

The above example motivates us to propose an approach which can accurately simulate the execution of Android applications, and identify complex wake lock misuse patterns.

3 NAVYDROID APPROACH

In this section, we elaborate on the approach that NavyDroid takes to detect energy inefficiency problems.

3.1 Overview

NavyDroid follows the two-part architecture described in Section 1. Its simulation part, named *application execution engine*, is integrated with JPF, a testing and verification framework for Java programs [8]. Its monitor part consists of a *misuse checker*, which checks for misuses of sensor listeners and wake locks, and a *sensor data utlization analyzer*, which evaluates the usage of sensor data and reports the cases of low utilization.

Figure 2 shows the high-level abstraction of NavyDroid. It takes an Android application as input, and produces an analysis report containing detected energy inefficiency bugs and corresponding program traces. The *application execution engine* simulates the execution of an application by executing it in JPF's Java virtual machine.

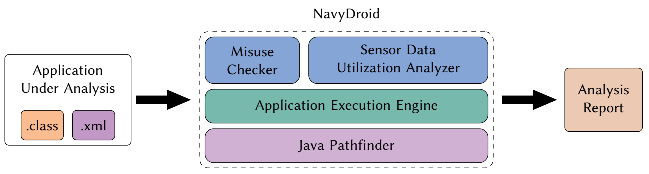

NavyDroid
Application Under Analysis
.class
.xml
Misuse Checker
Sensor Data Utilization Analyzer
Application Execution Engine
Java Pathfinder
Analysis Report

Figure 2: Approach overview

During the execution, the *misuse checker* monitors the registeration/unregistration of sensor listeners, and the acquisition/releasing of wake locks. The *sensor data utlization analyzer* feeds sensor data to the application when sensor data is requested, tracks the propagation of sensor data during the execution, and evaluates the utilization of sensor data at each application state. When the execution ends, NavyDroid reports the misuses of sensor listeners and wake locks, and the states where sensor data is underutilized. We detailedly present these parts of NavyDroid in the following.

3.2 Supporting Technology

JPF is a testing and verification framework for Java programs [8]. The core of JPF is a virtual machine for Java bytecode. We execute Android applications in JPF's virtual machine. In this way, we build NavyDroid on top of JPF, and employ JPF's facilities to implement the components of NavyDroid.

JPF is designed for analyzing traditional Java programs. It relies on explicit calling relationships in code. However, since Android applications follow event-driven mechanisms, the application logics of Android applications is separated in event handlers that are implicitly called at runtime. Hence, we guide JPF to schedule event handlers according to our application execution model.

We leverage JPF's *instruction listener* and *native peer* mechanisms to implement the functionalities of NavyDroid.

Instruction listener. Instruction listeners provide a way to observe the instruction execution [6]. By implementing several instruction listeners, NavyDroid is able to keep track of an Android application's execution, such as method invocations.

Native peer. A *native peer class* models a JVM's native class executed by JPF's virtual machine [7]. With native peers, we can simulate the Android framework APIs, and provide mock classes for some critical classes such as `LocationListener` and `WakeLock`.

3.3 Application Execution Engine

Different from traditional Java programs, Android applications follow an event-driven paradigm. The flow of the program is determined by events such as user actions or messages from the Android system. An application listens for events, and invokes corresponding event handlers when receiving an event.

The application execution engine simulates the execution of an Android application. It systematically generates user events and system events, and schedules the corresponding event handlers. The engine relies on an *application execution model* (AEM) to guide the runtime scheduling of event handlers.

3.3.1 Event Sequence Generation. In order to systematically explore an application's state space, the application execution engine generates event sequences to simulate user interactions and system messages.

Definition 3.1. An event sequence *seq* is an ordered list

$$seq = [e_1, e_2, \dots, e_n] \quad (e_i \in E, i = 1, 2, \dots, n)$$

where E denotes the set of events.

Before executing the application, NavyDroid first analyzes the configuration files to extract the GUI layouts of activities. Each GUI widget (e.g., a button) receives a set of user actions (e.g., button clicks). We take the union of these user actions and construct an *event set* for each activity. We also add into the event set some special events, including physical keys, namely *back*, *menu* and *home*, as well as system event *kill*.

During the execution, the application execution engine monitors all activities' states. When an activity stays at foreground and waits for user interactions, NavyDroid randomly picks an event from the *event set* and feeds it to the activity.

The application execution engine keeps generating events, until the execution of the application finishes, or the number of events sent to the application reaches an upper bound. [1] We add implicit *back* event when the event sequence generation stops while the application is not finished.

3.3.2 Event Handler Scheduling. With event sequences generated to represent user interactions, we now consider how to schedule event handlers properly. Scheduling event handlers is based on the event sequence and the AEM module.

[1] The length of event sequence is bounded so that the state space exploration can finish in finite time. Experiments show that a bounded state space is enough for NavyDroid to locate energy inefficiency bugs.

The application execution model (AEM) specifies how event handlers should be scheduled. Generally, an application execution model is an independent module, which takes an event as input, and returns a list of event handlers as output. It stores application state, and determines which event handlers to schedule when events come. In each iteration, the application execution engine consumes an event from the event sequence, consults the AEM about which event handlers should be scheduled, and invokes the event handlers orderly.

An application execution model can have various forms. GreenDroid defines AEM as a collection of temporal rules with the form of $[\psi], [\phi] \Rightarrow \lambda$, where ψ, ϕ and λ denote the execution history, the current situation, and what is expected to happen, separately [19]. E-GreenDroid represents AEM by an abstract state machine, because temporal rules are not practical in coding [25]. However, the AEM of E-GreenDroid is not precise enough to simulate the runtime behaviors of Android applications. Therefore we propose a new AEM for application exeuction. We define the structure of our AEM formally and describe its improvement as follows.

We construct the AEM as a strengthened DFA (deterministic finite automaton). We define a DFA with application states as its inner states, user events as its inputs, and enhance it with additional outputs of evnet handlers.

Definition 3.2. An application execution model *AEM* is a 7-tuple, $(S, E, H, \delta, f, s_0, F)$, consisting of

- a finite set of application states (S)
- a finite set of input events (E)
- a finite set of output event handler sequences (H)
- a transition function ($\delta : S \times E \longrightarrow S$)
- a scheduling function ($f : S \times E \longrightarrow H$)
- a start application state ($s_0 \in S$)
- a set of final application states ($F \subseteq S$)

Our AEM is different from a normal DFA in event handler sequences H and scheduling function f. If s is the current application state and e is the incoming user event, then $h = f(s, e) \in H$ is the list of event handlers to schedule at the current state. Figure 3 illustrates the transition diagram of our AEM model. What are marked on the edges are the event handlers scheduled during state transitions. The AEM relates closely to the lifecycle of an activity. For example, the AEM enters the *resumed* state when the activity comes to foreground and the `onResume()` callback is invoked; when the activity is switched to the background and becomes invisible, the `onPause()` and `onStop()` callbacks are invoked and the AEM enters the *stopped* state.

Compared with E-GreenDroid's state machine model, our AEM is more accurate when modeling the case of (1) an activity staying paused, and (2) an activity being killed.

The *paused* state is the intermediate state when an activity transits from *resumed* to *stopped.* An activity enters the *paused* state when a new, translucent activity (such as a dialog) opens, and remains paused as long as it is still partially visible but not in focus. An activity can be killed by the Android system when the memory is insufficient. The likelihood for an activity to be killed differs among states, as Table 1 shows [1].

In this way, we build an AEM that precisely specifies the runtime behaviors of Android applications. As the application execution

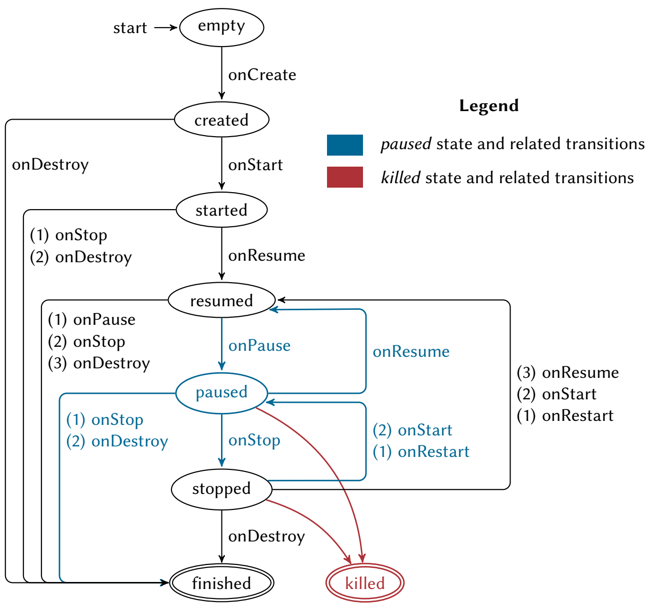


Figure 3: NavyDroid AEM model (the activity part)

Table 1: The correlation between activity state and the likelihood of the system's killing the process

Activity state	Likelihood of being killed
created, started, resumed	least
paused	more
stopped, destroyed	most

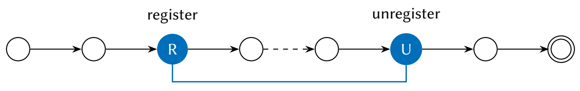
path is determined by the AEM, NavyDroid simulates the execution of an application more accurately, and better detect energy inefficiency problems with its precise AEM.

3.4 Sensor and Wake Lock Misuse Check

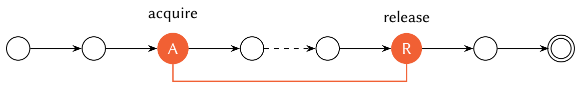
During the execution of an application, the *misuse checker* monitors the operations on sensor listeners and wake locks to check whether they are misused. As mentioned before, missing sensor listener or wake lock deactivation is one of the energy inefficiency patterns. Sensor listeners and wake locks share a similar concept as requests to system resources. A sensor listener is a request to sensor data, while a wake lock is a request for the CPU to stay awake. However, the behaviors of wake locks can be more complicated than sensor listeners. Here we explain misuse patterns of sensor listeners and wake locks, respectively.

Sensor listener misuse patterns. Figure 4a shows the use mode of sensor listeners. In all execution paths, a sensor listener should be unregistered when the sensor data is no longer needed. NavyDroid records the status of sensor listeners. It checks unregistered sensor listeners when each execution of the application finishes.

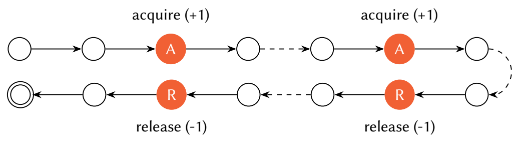
Wake lock misuse patterns. As Table 2 shows, there are eight misuse patterns of wake locks, and five of those patterns associate with energy waste, according to the empirical study in [20]. Existing



(a) Sensor listeners



(b) Wake locks (not reference counted)



(c) Wake locks (reference counted)

Figure 4: Use modes of sensor listeners and wake locks

work, such as GreenDroid and E-GreenDroid, can only deal with one pattern, i.e. *wake lock leakage*. NavyDroid also addresses the *multiple lock acquisition* pattern besides wake lock leakage.

The multiple lock acquisition pattern associates with the *reference-counted* mode of wake locks. When a wake lock is not reference counted, its behaviors are similar to that of a sensor listener, as Figure 4b shows. However, the behaviors of wake locks become more complicated with reference counts, as is shown in Figure 4c. An acquiring operation adds one to the counter, while a releasing operation subtracts one from the counter. A wake lock remains held until its count decreases to zero. Therefore, if a wake lock's reference count is larger than zero when the execution of an application ends, the phone will keep awake with no benefits to users, suggesting a wake lock misuse.

Misuse checker. NavyDroid uses the following polices to check for the misuses of sensor listeners and wake locks:

- A sensor listener *sl*, once registered, should be eventually unregistered when the execution of the application finishes.
- A wake lock *wl*, if not reference counted, once acquired, should be released at least once when the execution of the application finishes.
- A wake lock *wl*, if reference counted, once acquired, should be released as many times as acquired when the execution of the application finishes.

NavyDroid monitors the operations of sensor listeners using an instruction listener, and stores the status of sensor listeners. For wake locks, it uses a native peer class to model both modes of wake locks. The native peer class can also collect the detailed information, such as the level and flags, of wake locks. We also assign unique IDs for sensor listeners and wake locks to better monitor their behaviors.

3.5 Sensor Data Utilization Analysis

NavyDroid analyzes the utilization of sensor data during the simulation execution. As mentioned before, sensor data should be utilized effectively to bring user benefits, thus energy inefficiency problems occur when sensor data is underutilized. We do not analyze the

Table 2: Common patterns of wake lock misuses and existing tools' capability

Misuse pattern	# issues	# affected apps	Related to energy waste	E-GreenDroid solvable	NavyDroid solvable
Unnecessary wakeup	11	7	Yes	-	-
Wake lock leakage	10	7	Yes	Solvable	Solvable
Permature lock releasing	9	7	No	-	-
Multiple lock acquisition	8	3	Yes	-	Solvable
Inappropriate lock type	8	3	Yes	-	-
Problematic timeout setting	3	2	No	-	-
Inappropriate flags	2	2	Yes	-	-
Permission errors	2	2	No	-	-

utilization of wake locks, because wake locks only associate with power management, and do not obtain or utilize data like sensors.

In order to analyze the utilization of sensor data, NavyDroid tracks the flow and transformation of sensor data, and evaluates the utilization of sensor data at each application state. Generally speaking, the utilization analysis contains three phases:

(1) Tainting phase: marking each sensor data object with a unique tag.
(2) Propagation phase: propagating taint marks as the application executes.
(3) Analysis phase: evaluating sensor data utilization at each application state.

Tainting sensor data. NavyDroid generates sensor data from a data pool and feeds it to the application when the application requests for sensor data through sensor listeners. Each sensor data object is initialized with a unique taint mark before it is fed to the application.

Propagating taint marks. We propagate taint marks at the bytecode instruction level, following a collection of taint propagation rules [19]. The taint marks of two operands are merged to be the result's taint mark. We propagate taint marks using JPF's *object attribution* functionality.

Analyzing sensor data utilization. For each application state, we calculate its *data utilization coefficient* (DUC) as [18]:

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

The DUC of sensor data d at state s is defined as the ratio between the usage of d at state s and the maximum usage of any sensor data at any state. The usage of sensor data d at state s is defined as [18]:

$$usage(s, d) = \sum_{i \in instr(s,d)} weight(i, s) \times rel(i) \quad (2)$$

In Equation 2, $instr(s, d)$ denotes the set of bytecode instructions executed after d is fed to the application. Indicator function $rel(i)$ tests whether an instruction i uses any data with the same mark as d has. Function $weight(i, s)$ assigns a weight to instruction i to measure the benefits that i brings to user.

A low DUC value indicates a low utilization of sensor data at an application state. We consider states with DUC less than a threshold to be the signs of energy inefficiency bugs. In this way, through taint marking, taint propagation and DUC calculation, we analyze the utilization of sensor data at each application state, and identify the inefficiency in sensor data utilization.

4 EVALUATION

We implemented our approach as a prototype tool named NavyDroid. Figure 5 shows the user interface of this prototype tool. NavyDroid is extended from E-GreenDroid, as E-GreenDroid is a state-of-the-art energy inefficiency diagnosis tool. We rewrote the application execution model of E-GreenDroid to accurately simulate the paused state and the killed state. During simulation executions, NavyDroid adds implicit *back* events at the end of each event sequence to ensure that an execution of the application terminates after consuming all the events. We also implemented a native peer class of the `WakeLock` class to monitor the misuse patterns of wake locks.

In this section, we evaluate the effectiveness of our approach. On the one hand, we evaluate whether NavyDroid hold the same effectiveness as E-GreenDroid. On the other hand, we evaluate the enhancement of NavyDroid compared with E-GreenDroid. We aim to answer the following two research questions:

- **RQ1 (Ability Equivalence):** Does NavyDroid hold the abilities that E-GreenDroid does; i.e., can NavyDroid conduct effective analysis on those applications with energy inefficiency bugs that E-GreenDroid can effectively analyze?
- **RQ2 (Ability Enhancement):** Can NavyDroid detect energy inefficiency problems that E-GreenDroid is not able to detect?

4.1 Experimental Setup

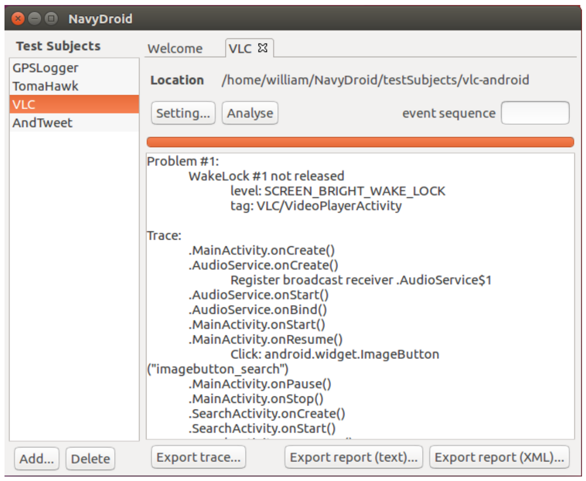
In order to answer RQ1, we selected all the 13 Android applications used as test subjects in E-GreenDroid's evaluation [25]. Table 3 lists the basic information of these 13 applications, which all have energy inefficiency bugs that can be detected by E-GreenDroid. For RQ2, we selected four real-world Android applications as test subjects. Table 4 lists the basic information of these applications. The last three applications have energy inefficiency problems of the multiple lock acquisition pattern according to the empirical study in [20]. We obtained these applications' source code and compiled them on Android 5.0 for our experiments.

We conducted our experiments on a dual-core computer with Intel core i5 CPU and 8GB RAM, running Ubuntu 16.04. We controlled E-GreenDroid and NavyDroid to generate 5,000 event sequences that do not exceed six in length. This length is enough for

Table 3: Information and analysis results of RQ1's test subjects

Application	Revision	Lines of code	E-GreenDroid results[1]	NavyDroid results[1]	Equivalent[2]
AndTweet	V-0.2.4	8,908	WLM	WLM	Yes
AAT	V-0.9-alpha	52,800	SDU	SDU	Yes
BabbleSink	R-d12879a	1,718	WLM	WLM	Yes
CWAC-Wakeful	R-d984b89	896	WLM	WLM	Yes
GPSLogger	R-15	659	SLM, SDU	SLM, SDU	Yes
GPSLogger-new	-	789	SLM, SDU	SLM, SDU	Yes
LocWriter2	V-0.1.1	1,542	SDU	SDU	Yes
OmniDroid	R-863	12,427	SDU	SDU	Yes
OsmDroid	R-750	18,091	SDU	SDU	Yes
Recycle Locator	R-68	3,241	SLM	SLM	Yes
RedBlackTree	R-0	483	WLM	WLM	Yes
Sofia Public Transport Nav.	R-114	1,443	SDU	SDU	Yes
Ushahidi	R-9d0aa75	10,186	SLM	SLM	Yes

[1] We denote **SLM** as *sensor listener misuse*, **WLM** as *wake lock misuse*, and **SDU** as *sensor data underutilization*.
[2] Whether E-GreenDroid and NavyDroid report the equivalent results.



Location /home/william/NavyDroid/testSubjects/vlc-android

```
Problem #1:
        WakeLock #1 not released
                level: SCREEN_BRIGHT_WAKE_LOCK
                tag: VLC/VideoPlayerActivity

Trace:
        .MainActivity.onCreate()
        .AudioService.onCreate()
                Register broadcast receiver .AudioService$1
        .AudioService.onStart()
        .AudioService.onBind()
        .MainActivity.onStart()
        .MainActivity.onResume()
                Click: android.widget.ImageButton
("imagebutton_search")
        .MainActivity.onPause()
        .MainActivity.onStop()
        .SearchActivity.onCreate()
        .SearchActivity.onStart()
```

Figure 5: The user interface of NavyDroid

E-GreenDroid and NavyDroid to explore a considerable application state space and detect energy inefficiency bugs.

For all test subjects, we ran both E-GreenDroid and NavyDroid to diagnose them, and examined the analysis reports to compare their ability to locate energy inefficiency bugs. In the following, we elaborate on our experiments with respect to the two research questions.

4.2 RQ1: Ability Equivalence

To answer RQ1 about the abilities of NavyDroid and E-GreenDroid being equivalent, we ran both E-GreenDroid and NavyDroid to analyze each application listed in Table 3. We collected the results of NavyDroid and E-GreenDroid for comparison.

Table 3 also shows the qualitative analyzing results of E-GreenDroid and NavyDroid. We classified energy inefficiency bugs into three categories, namely *sensor listener misuse*, *wake lock misuse* and *sensor data underutilization*. E-GreenDroid and NavyDroid explicitly report a sensor listener misuse or a wake lock misuse if they locate bugs in these types. For sensor data underutilization, E-GreenDroid and NavyDroid report all the application states with a DUC less than 1.0. We divided different severity levels of sensor data underutilization according to the DUC. The severity level of a problem is *severe* if the DUC is less than 0.5, is *mild* if the DUC is less than 0.8 but no less than 0.5, and is *low* if the DUC is no less than 0.8. We considered that a bug occurred when the underutilization was *severe* and classified the bug as *sensor data underutilization*. In this way, NavyDroid reported the same categories of energy inefficiency bugs as E-GreenDroid did.

We compared the detailed information in E-GreenDroid's and NavyDroid's reports in order to further demonstrate the effectiveness of NavyDroid. For energy inefficiency bugs classified as *sensor listener misuse* and *wake lock misuse*, we examined whether the misuse patterns were the same, as NavyDroid can report two types of misuse patterns of wake locks. Moreover, we compared the execution traces of each reported energy inefficiency bug. We considered the effectiveness of E-GreenDroid and NavyDroid to be equivalent when they reported bugs with the same misuse patterns and the same execution traces. [2] For energy inefficiency bugs classified as *sensor data underutilization*, we examined whether the problematic states had the same severity level of underutilization. Furthermore, we compared the DUC levels across the states from both reports. Figure 6a, 6b, 6c and 6d shows the overview of DUC levels of GPSLogger and OsmDroid analyzed by E-GreenDroid and NavyDroid. We can see from the bar chart that the distributions of DUC values reported by E-GreenDroid and NavyDroid are basically the same. Both of them reports sensor data underutilization of the *severe* level. In summary, NavyDroid performs effective analysis and can locate all the energy inefficiency bugs that E-GreenDroid locates.

[2] The execution traces recorded by E-GreenDroid and NavyDroid cannot be exactly the same because NavyDroid simplifies and refactors the traces. We only examined whether the key callbacks of components were the same.

Table 4: Information and analysis results of RQ2's test subjects

Application	Revision	Lines of code	E-GreenDroid results[1]	NavyDroid results[1]	Improved[2]	Cause[3]
AndroidRun	R-dbf6428	1,649	SLM[4]	SLM	Yes	AEM
VLC	R-abe60f5	6,839	NPD	WLM	Yes	MLA
CSipSimple[5]	R-152	13,107	NPD	NPD	No	-
TomaHawk	R-543c3b9	4,601	NPD	WLM[6]	Yes	AEM, MLA

[1] We denote **NPD** as *no problem detected*, **SLM** as *sensor listener misuse*, and **WLM** as *wake lock misuse*.
[2] Whether the results of NavyDroid is better than E-GreenDroid.
[3] Why NavyDroid reports better analyzing results than E-GreenDroid. **AEM** represents the improvement in application execution model, and **MLA** represents the ability of monitoring the misuse pattern of multiple lock acquisition.
[4] The SLM bug that E-GreenDroid reports in AndroidRun is a false positive. We will discuss this in detail.
[5] CSipSimple has six problematic revisions according to the empirical study. We compress these six revisions because the results are all NPD.
[6] NavyDroid detects two different WLM bugs in TomaHawk.

4.3 RQ2: Ability Enhancement

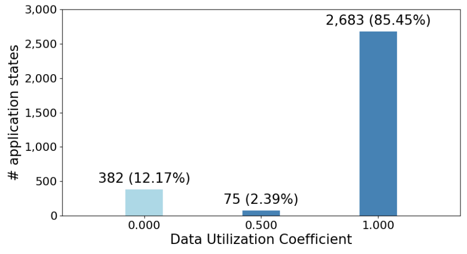
To answer RQ2 about the improvement in abilities of NavyDroid compared with E-GreenDroid, we selected four open-source Android applications as test subjects. Table 4 presents the basic information and analyzing results of these test subjects. We collected the results of E-GreenDroid and NavyDroid, and classified energy inefficiency bugs in the same way as Section 4.2. E-GreenDroid failed to detect any energy inefficiency bugs for three out of four test subjects, and the only case that E-GreenDroid reports an energy inefficiency bug was a false positive. Whereas, NavyDroid reported four energy inefficiency bugs in three applications. We discuss these four test subjects and the detected problems as follows.

AndroidRun. AndroidRun is a location tracking and speed calculating application for running and biking [3]. NavyDroid reported that a location listener (a kind of sensor listeners) is not unregistered. AndroidRun uses a location listener to collect GPS data in the main activity. The location listener is registered in the `onCreate()` callback and unregistered in the `onDestroy()` callback. If the activity is switched to the background and killed by the Android system, the `onDestroy()` callback will not be invoked, and the location listener will never be unregistered. E-GreenDroid also reported this sensor listener misuse. However, E-GreenDroid checks for the misuse of sensor listeners immediately after all the events have been sent to the application and all the event handlers have been scheduled, when the activities have not finished their lifecycles yet. We considered the reported sensor listener misuse as a false positive in this case. In NavyDroid's implementation, we avoided this false positive by checking for sensor listener misuse after all the activities and services are finished. NavyDroid can still reported the misuse of the location listener because it simulates the kill event sent by the Android system.

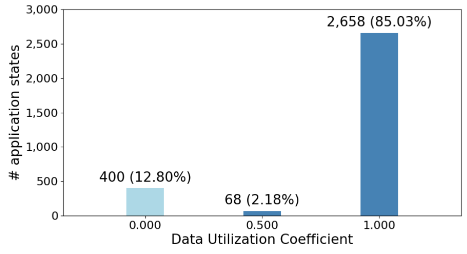
VLC. VLC is an open source media player and framework [12]. NavyDroid reported that in certain cases the application acquires a wake lock more than once but the reference count is not decreased to zero when the application terminates. The `VideoPlayerActivity` is responsible for playing the video specified by users. It employs a wake lock to keep the screen on while playing the video. Users click the play/pause button to switch the mode of video playing. When a user plays the video, the wake lock is acquired; and when a user pauses the video, the wake lock is released. However, the wake lock is also acquired when the `VideoPlayerActivity` is resumed and the media file is loaded. If a user navigates to this activity and clicks the button twice, the wake lock will not be released, and the screen will stay on, wasting battery power. This energy inefficiency bug matches the multiple lock acquisition pattern of wake lock misuse, which NavyDroid can effectively analysis. E-GreenDroid failed to locate this energy inefficiency bug because it does not store the reference counts of wake locks, and considers a wake lock to be released once `release()` is invoked.

CSipSimple. CSipSimple is a SIP (Session Initiation Protocol) application for communicating over the Internet [5]. Recent empirical study [20] found wake lock misuses of the multiple lock acquisition pattern in CSipSimple. However, both E-GreenDroid and NavyDroid detected no energy inefficiency problems in it. CSipSimple maintains a `SipService` in the background to register accounts. If some accounts succeed in registration, a wake lock is acquired to keep the screen on. The call to `acquire()` of the wake lock can be repeated and there may not be an equal number of calls to `release()`. Nevertheless, the wake lock object is set to non-reference counted when created, and one call to `release()` is sufficient to undo the effect of all previous call to `acquire()`. So we considered that CSipSimple has no wake lock misuse in this case, and NavyDroid reported a reasonable analysis result.

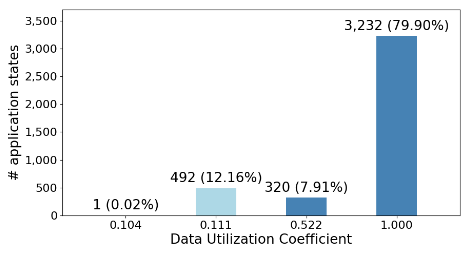
TomaHawk. TomaHawk is a multi-source music player [11]. NavyDroid reported two energy inefficiency bugs classified as *wake lock misuse.* As is described in the motivating example in Section 2.2, the two bugs occurs in different cases. If a user clicks the play/pause button twice after the media player gets prepared, the wake lock is acquired twice but released once, and remains held as its reference count is larger than zero. E-GreenDroid fails to locate the energy inefficiency bug in this case because of its inability to analyze the multiple lock acquisition pattern. Another bug occurs when the Android system kills the activity and the service associated with media playing, in which case the `onDestroy()` callback is not invoked, causing wake lock leakage. Because of its imprecise application execution model, E-GreenDroid cannot simulate the behaviors of



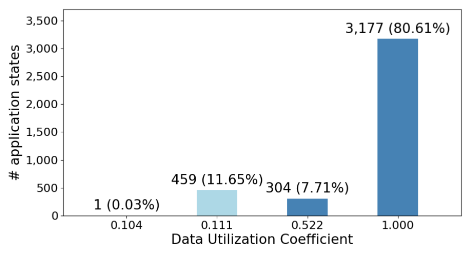
(a) DUC of GPSLogger analyzed by E-GreenDroid



(b) DUC of GPSLogger analyzed by NavyDroid



(c) DUC of OsmDroid analyzed by E-GreenDroid



(d) DUC of OsmDroid analyzed by NavyDroid

Figure 6: DUC distribution bar charts

the application with the kill event, and fails to locate this energy inefficiency bug.

4.4 Discussion

Like E-GreenDroid, NavyDroid is also implemented on top of JPF. Simulating the execution of Android applications by JPF has shortcomings in modeling the Android framework APIs. We choose JPF as the underlying analyzing framework because we can monitor and intercept the execution of applications. In our approach, the enhanced AEM guides the simulation execution of Android applications. It stores the state of the application, and schedules event handler according to the current state. Our AEM may still be inconsistent with the actual execution of the application in some cases. However, it is highly extensible and can be easily updated.

Currently, NavyDroid can detect two misuse patterns of wake locks. NavyDroid cannot address the *unnecessary wakeup* pattern, which is the most common pattern of wake lock misuse according to the empirical study [20]. We will extend NavyDroid to analyze this pattern in the future. Wake locks have some other misuse patterns such as *inappropriate lock type* and *inappropriate flags*. However, these two patterns lack feasible criteria, thus they are difficult to be detected automatically.

5 RELATED WORK

Our work relates to existing studies of several research topics, which includes energy inefficiency analysis and wake lock misuse detection. In this section, we discuss some representative pieces of work in recent years.

Energy efficiency analysis. Researchers have proposed various approaches that detect energy inefficiency bugs in smartphone applications. Pathak et al. characterized no-sleep energy bugs in Android apps, and detected no-sleep energy bugs using reaching-definition data-flow analysis [23]. Liu et al. revealed energy problems caused by sensor data underutilization, and proposed GreenDroid to locate no-sleep bugs by adapting resource leak detection and dynamic dataflow analysis [19]. This work formulated the scheduling policies of event handlers, thus GreenDroid requires no information about control flows among event handlers. ADEL is a dynamic taint-tracking analysis infrastructure to detect energy leaks of network data [27], and is similar to GreenDroid in locating energy bugs caused by inefficient use of program data. Li et al. built CyanDroid to diagnose sensor data underutilization, which can systematically explore an application's state space by generating multi-dimensional sensor data using the white-box sampling technique [17]. Wang et al. updated and optimized GreenDroid to support new Android features, and abstracted a state machine for event handler scheduling.

Some work repairs energy inefficiency bugs besides detecting them. Ma et al. presented a practical tool that identifies the abnormal behaviors of an application, and suggests the most approapriate repair solution to users [21]. Banerjee et al. developed EnergyPatch, a framework that combines static and dynamic analysis technique to detect, validate and repair energy bugs in Android applications [13].

There are also some existing studies concerning energy consumption estimation. Estimating the energy consumption of an application can show developers which components consume the most energy. Pathak et al. implemented Eprof to measure energy consumption of Android applications by estimating the power states of components during system calls [22]. Besides Eprof, several studies [15, 16, 28] also estimate the energy consumption and identify energy hotspots and energy bugs of Android apps. Both eLens [15] and vLens [16] can evaluate fine-grained energy consumption information at a source-line level. Banerjee et al. proposed a framework that automatically generate test inputs to detect energy bugs and energy hotspots [14]. Energy consumption estimation approaches identify energy hotspots in smartphone applications. However, all energy hotspots are not energy inefficiency bugs. An energy hotspot is an energy inefficiency bug when the energy consumption is not necessary and produces no user benefits.

Wake lock misuse detection. Misuses of wake locks can crash applications or cause severe energy problems. Pathak et al. employed dataflow analysis to locate no-sleep bugs caused by wake lock leakage [23]. Vekris et al. proposed a static analysis tool that verifies the absence of wake lock misuses with respect to a set of policies [24]. GreenDroid and E-GreenDroid can monitor the API calls of wake locks during simulation execution to detect wake lock leakage [19, 25]. Wang et al. implemeneted WLCleaner to repair wake lock issues at runtime [26]. However, these studies focused on energy inefficiency bugs caused by wake lock leakage. Liu et al. conducted an empirical study to understand common wake lock usage in practice, and summarized eight common patterns of wake lock misuses [20].

6 CONCLUSION

In this paper, we have presented an approach that detects energy inefficiency problems in Android applications. Our approach simulates the runtime behaviors of an application with a precise application execution model derived from Android specifications. During the execution, it monitors the usage of sensor listeners and wake locks, and the utilization of sensor data. It can recognize wake lock misuses that match the multiple lock acquisition pattern. We implemented our approach on top of JPF, and evaluate it with real-world applications. The results demonstrate its effectiveness in diagnosing energy inefficiency problems.

In the future, we plan to further extend this work to support more patterns of energy inefficiency problems. We also plan to re-implement our tool on top of Android virtual machine. At present, dynamic analysis tools can only simulate the execution of Android applications and may be inconsistent with the actual execution of applications. The analysis can be more precise by inspecting the real execution on Android virtual machine.

ACKNOWLEDGMENTS

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 61690204) of China. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] 2017. Android Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html. (2017).

[2] 2017. Android Application Fundamentals. https://developer.android.com/guide/components/fundamentals.html. (2017).

[3] 2017. Android Run. https://sourceforge.net/projects/androidrun/. (2017).

[4] 2017. Android Sensors Usage. https://developer.android.com/guide/topics/sensors/sensors_overview.html. (2017).

[5] 2017. CSipSimple. https://github.com/r3gis3r/CSipSimple. (2017).

[6] 2017. Java Pathfinder Listeners Wiki Page. https://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/listener. (2017).

[7] 2017. Java Pathfinder MJI Wiki Page. https://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/mji. (2017).

[8] 2017. Java Pathfinder Wiki Page. https://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/what_is_jpf. (2017).

[9] 2017. Managing Android Device Awake State. https://developer.android.com/training/scheduling/wakelock.html. (2017).

[10] 2017. PowerManager.WakeLock Class. https://developer.android.com/reference/android/os/PowerManager.WakeLock.html. (2017).

[11] 2017. Tomahawk. https://github.com/tomahawk-player/tomahawk-android. (2017).

[12] 2017. VLC. https://github.com/mstorsjo/vlc-android. (2017).

[13] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2017. Energypatch: Repairing Resource Leaks to Improve Energy-efficiency of Android Apps. *IEEE Transactions on Software Engineering* (2017).

[14] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, 588–598.

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

[16] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. 2013. Calculating Source Line Level Energy Information for Android Applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA '13)*. ACM, 78–89.

[17] Qiwei Li, Chang Xu, Yepang Liu, Chun Cao, Xiaoxing Ma, and Jian Lü. 2017. CyanDroid: Stable and Effective Energy Inefficiency Diagnosis for Android Apps. *Science China Information Sciences* 60 (2017), 012104.

[18] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2013. Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. In *IEEE International Conference on Pervasive Computing and Communications (PerCom'13)*. IEEE, 2–10.

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

[20] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and Detecting Wake Lock Misuses for Android Applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '16)*. ACM, 396–409.

[21] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. 2013. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, 57–70.

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

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

[24] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. 2012. Towards Verifying Android Apps for the Absence of No-sleep Energy Bugs. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. USENIX Association.

[25] Jue Wang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lü. 2016. E-GreenDroid: Effective Energy Inefficiency Analysis for Android Applications. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware*. ACM, 71–80.

[26] Xigui Wang, Xianfeng Li, and Wen Wen. 2014. Wlcleaner: Reducing Energy Waste Caused by Wakelock Bugs at Runtime. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC '14)*. IEEE, 429–434.

[27] Lide Zhang, Mark S. Gordon, Robert P. Dick, Z. Morley Mao, Peter Dinda, and Lei Yang. 2012. ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '12)*. ACM, 363–372.

[28] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '10)*. ACM, 105–114.