Source-Code Based Energy Anomalies Diagnosis Algorithm for Android Applications

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By
Yun Li

Graduate Program in Electrical and Computer Science

The Ohio State University
2015

Master's Examination Committee:
Prof. Xiaorui Wang, Advisor
Prof. Feng Qin
Abstract

Smart phone marketplace grows fast on recent years, which allows a large number of developers freely release tons of thousands apps every day. However, since developers’ experience vary from novice to expert, the apps quality and code efficiency cannot be guaranteed. Some developers can write clean and bug free apps, but quite a lot applications contain a certain kind of bug. Therefore, developers are in need of a tool for detecting bugs and fix them, which helps developers improve the quality of applications.

Also, smart phone has a limited battery life, which makes energy-saving an essential topic for application developers. This feature lead to the mobile phone’s operating system’s aggressive power management policy and also lead to the requirement for programmers of avoiding misuse of power related API to conserve energy.

In this thesis, we proposed an algorithm for efficiently analyzing energy anomalies, pinpointing potential energy bug through source-based analyze. We tested our algorithm towards a number of open-source apps and analyze three typical cases to demonstrate the efficiency and accuracy of our algorithm. Further, we discussed future works regarding this topic.
Dedication

This document is dedicated to my friends.
Vita

2010 to 2014 ........................................ B.S. Electrical and Computer Engineer,

Shanghai University of Electric Power

2014 to present ..................................... M.S. Electrical and Computer Engineer, The

Ohio State University

Fields of Study

Major Field: Electrical and Computer Engineering
# Table of Contents

Abstract......................................................................................................................... ii

Dedication .................................................................................................................... iii

Vita................................................................................................................................. iv

List of Tables ................................................................................................................ vi

List of Figures .............................................................................................................. vii

Chapter 1: Introduction .............................................................................................. 1

Chapter 2: Background .............................................................................................. 3

Chapter 3: Approach .................................................................................................. 4

Chapter 4. Case Study ................................................................................................. 10

4.1 A better camera .................................................................................................... 10

4.2 App alarm ............................................................................................................. 12

4.3 congress-android .................................................................................................. 13

4.4 Pedometer ........................................................................................................... 15

4.5 Binaural-Beats .................................................................................................. 16

Reference .................................................................................................................... 18


List of Tables

Table 1. Levels of Power Wakelock ........................................................................ 5
Table 2. A Typical Usage of Power Wakelock ......................................................... 5
Table 3. Energy Bug Code Path ............................................................................. 6
Table 4 Code Lines of A Better Camera ................................................................. 11
Table 5. Code Lines of App Alarm ......................................................................... 13
Table 6. Code Lines of Congress-Android ................................................................ 14
Table 7. Code lines of Pedometer ......................................................................... 15
List of Figures

Figure 1. Life-cycle for android application.................................................................................. 8
Figure 2. CPU frequency and screen utilization ............................................................................. 16
Figure 3. Combined results of multiple users.................................................................................. 17
Chapter 1: Introduction

Smart phone marketplace grows fast on recent years, smart phone has surpassed desktop machine in sales in 2011 and has become the most popular computing platforms [1]. It became prevalent because of the feature of multi-cores, digital camera, large screen and GPS location. Smart phone marketplace allows a large number of creative and novel applications appear every day. However, since developers’ experience vary from novice to expert, the apps quality and code efficiency cannot be guaranteed. Some developers can write clean and bug free apps, but quite a lot applications contain a certain kind of bug. Therefore, developers are in urgent need of a tool for detecting bugs and fix them, which helps them improve the quality of apps.

Prior research work on energy bug can be categorized into two groups: 1) Source-based methods that analyze the application’s source code. 2) Trace-based methods that diagnose user phone traces.

Existing work on trace-based methods mainly concentrate on helping users to pinpoint energy anomalies, an example is that eDoctor [2] can tell users which app consume abnormal amount of energy through clustering executing application of user’s mobile phone into different phases according to their utility resources such as CPU, Wi-Fi and
screen. However, they are insufficient for developers to get a knowledge of which part of their application code have problems since eDoctor can only supply coarse-grained data.

On the other hand, source-based methods are powerful and can detect bugs that rarely manifest. Since source-based method can look deep into the whole program’s code, it can supply more detailed information than trace-based method for developers. Also, it is more efficient in giving the root-cause of possible energy bugs.

In this paper, we proposed a novel algorithm for energy anomalies diagnosis on smart phone applications, which can be categorized into source-based method. The paper makes the following major contributions:

1. We proposed a novel algorithm which can help developers efficiently catch energy bugs.
2. We implemented the algorithm through simple steps.

This paper is organized as following: the first chapter gives an introduction and overview for the topic, also discussed about previous literature. The second chapter provides background technology related to this topic. The third chapter detailed discusses the approach of the algorithm. The fourth chapter we shows the result and analyze several case to demonstrate the efficiency and accuracy of our approach.
Chapter 2: Background

An android application is composed from a set of components, which include four major elements: activity, service, broadcastReceiver and contentProvider. All of these components are associated with lifecycle of which the stages are corresponding to different parts of their functionality.

Energy bugs usually appear with hardware component usage, the list of hardware component can briefly fall into two categorize: 1) traditional component which include CPU, display, WIFI, memory and storage, 3D or 4D radio, these components can also be found in desktop computers. 2) External components which include GPS, camera and sensors. The android system supports three kinds of sensors [3]: a) motion sensors which measure acceleration forces and rotational forces along three axes, such as accelerometers, gravity sensors, gyroscopes, and rotational vector sensors. b) Environmental sensors which measure various environmental parameters, like ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers. c) Position sensors which measure the physical position of a device such as orientation sensors and magnetometers.
Chapter 3: Approach

Unlike desktop computers, smart phone has a limited battery life. Therefore, smart phone gives power management policy a high priority, namely every component, including the CPU, stays off or in an idle state, unless the app explicitly instruct the operating system to keep on. All smart systems such as Android, IOS employ a strictly power management policy, putting all of the component into sleep or into an idle state immediately after users inactivity of a small period. The smart phone as a whole draws nearly zero power in an idle state. This policy gives a relative long battery life, usually smart phone can last dozens of hours when all the elements are in an idle state.

Because of the aggressive power management policy of smart phone, developers need to use certain instructions to keep the operating system on when they want to implement important function where interruption may cause a severe failure or disruption of application. Specifically, developers need to use new WakeLock() API to explicitly instruct operating system to keep on, and use methods of PowerManager.WakeLock to control the power state of devices. Wake lock levels are defined with varying effect on operating system. Table 1 shows four levels of power wakelock.
<table>
<thead>
<tr>
<th>Flag Value</th>
<th>CPU</th>
<th>Screen</th>
<th>Keyboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL_WAKE_LOCK</td>
<td>On</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>SCREEN_DIM_WAKE_LOCK</td>
<td>On</td>
<td>Dim</td>
<td>Off</td>
</tr>
<tr>
<td>SCREEN_BRIGHT_WAKE_LOCK</td>
<td>On</td>
<td>Bright</td>
<td>Off</td>
</tr>
<tr>
<td>FULL_WAKE_LOCK</td>
<td>On</td>
<td>Bright</td>
<td>Bright</td>
</tr>
</tbody>
</table>

Table 1. Levels of Power Wakelock

In practical usage, developers should use minimum level possible to reduce power loss.

Table 2 is a typical usage of wakelock to control the power state.

```java
PowerManager pm = (PowerManager)
        getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl =
    pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "My Tag");
wl.acquire();
    ..screen will stay on during this section..
wl.release();
```

Table 2. A Typical Usage of Power Wakelock

When power wakelock is acquired in the program, it will make operating system keep active until released, which gives a chance for energy bugs in applications for three reasons.
First, programmers may simply forget to release power wakelock through the code or programmers may release wakelock on the if branch but forget to doing this in the else branch.

Second, although programmers did release wakelock in many code paths but the program took an unexpected code path during real time execution along which release of wakelock has not been put into code. This situation usually happens in try and catch.

Table 3 shows an example of energy bugs code path.

```
try{
    wl.acquire(); //CPU should not sleep
    net_sync(); //Throws Exception(s)
    wl.release(); //CPU is free to sleep
}

catch(Exception e) {
    System.out.println(e); //Print the error
}

finally {
} //End try-catch block
```

Table 3. Energy Bug Code Path

It represents a typical template of energy bug in which an app has taken a unexpected path. As shown in listing 2, net_sync() represents some task which cannot be interrupted, therefore programmers make it protected by wakelock acquire and release after executing. CPU will not go to sleep when executing net_sync(). However, routine
net_sync() may throw exceptions. Which is a mechanism for notifying some type of failure for developers, such as opening a file which don’t exist or dividing zero. A thrown exception can be caught by try-catch block. Then the code will go to the catch branch and skip the wakelock release, making code path uncontrollable and generate energy bug. Actually, wakelock release should be located in finally block which will always be executed.

Third, the most common pattern of energy bug code path is because of developer’s ignorance of life-cycle of android processes. Figure 1 shows a flow diagram of android application.
Figure 1 Life-cycle for android application
A fundamental feature of android system is that the lifetime of an application is not simply controlled by itself. Instead, it is determined by the system and affected by how important this application is to the users and how much overall memory is available in the system. Android system will make an "importance hierarchy" for each process to determine which process to be killed when memory is limited.

Another feature is that an android application will keep active once start, until RAM reaches its limit or the application kills itself. When user exit the application, android system saves apps’ state and pop it back when users go back to this application. It means when application calls onPause() Method, it is still alive but is not visible for users, this application keeps running of battery even if on the background. Therefore, if developers just release wakelock in onDestroy() method but not in onPause() method, the application will cost a large amount of power since OnDestroy() method will only be called when the application is completely killed or destroyed.

The energy bug code path discussed above use a wakelock as an example but can be similarly extend to any hardware component’s acquire and release.

When we have a knowledge of all the possible code paths for energy bug, it will become a trivial task to catch these bugs, developers can simply search for every acquire keyword and then check whether release keyword appears in the right place.
Chapter 4. Case Study

For testing, we did experiments on several apps. Through adding log before or after each acquire and release point in the code, we verified the efficiency and accuracy of our algorithm.

Since our method is source-based, we test our algorithm on several open-source applications. In this section, we analyze our case studies on five apps: “A better camera”, “App Alarm”, “congress-android”, “Pedometer” and “Binaural-Beats”.

4.1 A better camera

A better camera contains a typical and common energy bug with a code path falls into the second classification. Table 4 shows a piece of code lines of a better camera’s AlarmReceiver class.
public void onReceive(Context context, Intent intent) {
    PowerManager pm = (PowerManager)context.getSystemService(Context.POWER_SERVICE);
    wakeLock = pm.newWakeLock(PowerManager.FULL_WAKE_LOCK | PowerManager.ACQUIRE_CAUSES_WAKEUP | PowerManager.ON_AFTER_RELEASE, TAG);
    wakeLock.acquire();
    try {
        if (ApplicationScreen.instance == null || ApplicationScreen.getCameraController() == null || (CameraController.isUseCamera2() ? CameraController.getCamera2() == null : CameraController.getCamera() == null)) {
            Intent dialogIntent = new Intent(context, MainScreen.class);
            dialogIntent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT | Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_SINGLE_TOP);
            context.startActivity(dialogIntent);
            wakeLock.release();
            takePicture();
        } catch (NullPointerException e) {
        }
    }
}

Table 4 Code Lines of A Better Camera
As we can see from table 4, the wakelock is acquired when the onReceive() method executed, but the developer release the power wakelock in the try block of try-catch, instead of releasing within finally block. Therefore, if exception is thrown in the try block, the release method will be missed on the real time execution. To verify, we added two logs immediately after wakelock acquire and release. Using eclipse to get runtime log data while playing with the app, we find there is a situation where wakelock is acquired but not released. Since it acquires a full wakelock, the screen is kept on although we have exited the app.

4.2 App alarm

App alarm contains an energy bug which can be categorized into the first classification. The programmer simply forgot release the wakelock throughout the whole code. Table 5 shows a piece of code lines of app alarm where wakelock is acquired but never released.
Table 5. Code Lines of App Alarm

From table 5 we can see that wakelock is acquired in SnoozeWakeupReceiver class but never released.

4.3 congress-android

congress-android contains an energy bug which can be categorized into the third classification. Table 6 shows a piece of congress-android which includes the bug code lines.
public int onStartCommand(Intent intent, int flags, int startId) {
    PowerManager.WakeLock lock=getLock(this.getApplicationContext());
    if (!lock.isHeld() || (flags & START_FLAG_REDELIVERY) != 0) {
        lock.acquire();
    }
    super.onStartCommand(intent, flags, startId);
    return(START_REDELIVER_INTENT);
}

@Override
final protected void onHandleIntent(Intent intent) {
    try {
        doWakefulWork(intent);
    }
    finally {
        PowerManager.WakeLock lock=getLock(this.getApplicationContext());
        if (lock.isHeld()) {
            lock.release();
        }
    }
}

Table 6. Code Lines of Congress-Android

According to the android application’s life cycle, if one acquired wakelock within the
onStartCommand() method, it has to release within that method to avoid an energy bug.
However, from table 6 we can find the programmer fails to do this properly.
4.4 Pedometer

Pedometer is an application for recording walking steps. The energy bug we found in this application is a typical one which can be categorized into the third code path of energy bug. Table 7 shows a piece of code lines of pedometer.

```java
private BroadcastReceiver mReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {

        if (intent.getAction().equals(Intent.ACTION_SCREEN_OFF)) {

            StepService.this.unregisterDetector();
            StepService.this.registerDetector();
            if (mPedometerSettings.wakeAggressively()) {
                wakeLock.release();
                acquireWakeLock();
            }
        }
    }
};
```

Table 7. Code lines of Pedometer

As shown in table 7, the programmer failed to release wakeLock with the onReceive method, which will invoke energy bugs during real time execution.
4.5 Binaural-Beats

Binaural-beats is an application that stimulates your brain by sending special auditory artifacts directly into your headphones to produce subtle changes in behavior through entrainment of brainwaves frequency.

This application failed to release wakelock within Onpause method and only release wakelock within OnDestroy method, which gives a high chance of entering the energy bug when users exit this application but RAM doesn’t reach its limit.

For verification, we analyzed the CPU frequency and screen usage through the running time of targeted applications, as shown in the figure 2 below:

![Figure 2. CPU frequency and screen utilization](image-url)
As shown in the figure, the red line represents CPU frequency versus time, and the blue line represents screen brightness versus time. Although the user have exited the application on point 40 minutes, the screen still is kept on and CPU frequency is abnormal, which indicates an energy bug. Figure3 shows a combined results of multiple users, which indicates a general situation.

Figure 3. Combined results of multiple users
Reference


