MAUE: A Framework For Detecting Energy Bugs From User Interactions On Mobile Applications

by

Menwa Alshammeri

B.S., Kuwait University, 2004

A Master Thesis submitted to the Graduate Faculty of the University of Colorado Colorado Springs in partial fulfillment of the requirements for the degree of Master of Engineering in Software Engineering Department of Computer Science 2015
This thesis for the Master of Engineering by
Menwa Alshammeri
has been approved for the
Department of Computer Science
by

Dr. Kristen Walcott-Justice, Chair

Dr. Edward Chow

Dr. Albert Glock

Date
Alshammeri, Menwa (M.E., Software Engineering)
A Framework For Detecting Energy Bugs From User Interactions On Mobile Applications
Thesis directed by Associate Professor Kristen Walcott-Justice

Smart phones are becoming ubiquitous in our society, being used for email, business, entertainment, every day activities, and more. Such devices are improving exponentially in terms of computation power and application complexity. Moreover, modern smartphones are equipped with a wide range of sensors and I/O components, such as GPS, WiFi, camera, and so on. However, the battery development does not correspond to the need for higher power consumption. Additionally, malfunctioning applications usually consume a higher amount of power than expected and drain the battery in a short time. Indeed, it is important to develop tools and techniques that aid in energy-efficient application development.

In this work, we aim at energy inefficiencies in smart-phones, namely energy bugs, caused by mobile applications. We propose a framework to dynamically detect such bugs on mobile applications, and we induce them to show up by interacting with the app. In order to do that, we monitor power consumption while running a sequence of user interactions that may lead to a potential energy bug in the application. We track power consumption and capture user actions using online trackers. We then analyze the the power data and runtime logs and report the resultant power breakdowns after mapping them to the corresponding sequence of actions. Finally, we report sequences of user actions and the associated power consumption, and identify the sequence that caused the energy bug
to manifest. Our experimental results show that our framework can dynamically
detect energy bugs caused by mobile applications and figure out what a user has
done to trigger them. This will help application developers identify where and
how the abnormal battery drain issues have occurred and determine regions in the
code that need investigation.
Acknowledgements

Firstly, I would like to thank my advisor Prof. Justice for her continuous support during my master study, for her patience, caring, and mentorship. Her guidance helped me in my research and writing of this thesis. My sincere thanks also goes to the rest of my thesis committee: Dr. Edward Chow, and Dr. Albert Glock, for their valuable comments and encouragement. Their questions and suggestions motivated me to widen my research and improve my work by developing an insightful look at the things from different prospectives. Last but not the least, I would like to thank my loved ones, who were always there for me. I’m grateful for my parents, my husband, my children and my brothers and sisters for the unceasing encouragement and support throughout writing this thesis and my life in general.
# TABLE OF CONTENTS

1 Introduction .................................................................................................................. 1

2 Background ..................................................................................................................... 5
   2.1 Energy Inefficiencies In Mobile Applications ......................................................... 5
   2.2 Energy Profiling in Mobile Devices ......................................................................... 8

3 Framework Design and Implementation ....................................................................... 11
   3.1 Framework Overview ......................................................................................... 12
   3.2 Estimating Power Consumption .......................................................................... 13
   3.3 General Background .......................................................................................... 14
      3.3.1 Android Framework .................................................................................. 15
      3.3.2 Aspect Oriented Programming .................................................................. 16
   3.4 Online-Tracking Component .............................................................................. 18
   3.5 Actions Tracker ................................................................................................. 20
   3.6 Offline Analysis .................................................................................................... 22

4 Evaluation ..................................................................................................................... 24
   4.1 Experimental Setup ............................................................................................. 24
   4.2 Integrating AspectJ Within Android .................................................................... 25
   4.3 Aspect Code ......................................................................................................... 27
   4.4 Subject Applications ......................................................................................... 28
5 Related Work

5.1 Energy Profiling on Mobile Devices ........................................ 42
5.2 Detecting Energy Bugs ............................................................ 43
5.3 Performance Testing ............................................................... 45
5.4 Debugging and Optimization for smart phone applications perfor-
    mance ...................................................................................... 45
5.5 Testing using Aspect-Oriented Programming ............................ 46

6 Conclusion and Future Work ......................................................... 47

Bibliography ................................................................................ 49
Chapter 1

Introduction

The mobile market has exponentially grown over the last few years. Recent estimations indicate that by 2015, over 70% of all handset devices will be smart-phones, capable of running mobile apps [1]. Today’s smart devices are being used to host a diverse set of applications due to the dramatic improvement in terms of processing speed and memory capacity. As more applications are ported to and created for mobile devices, application complexity also increases, and the quality of these programs becomes more critical.

Additionally, modern smart phones are equipped with a wide range of sensors and I/O components, such as GPS, WiFi, camera, and so on. These I/O components allow developers to create a diverse set of applications. However, the battery development is much slower than the hardware advances [2], and therefore, the battery life has become the most critical resource in smart phones.

Pathak et al. [3] have shown that the severity of the abnormal battery drain issues caused by various reasons, including hardware (22.93%), software (35.10%), and other external causes. They refer to such abnormal battery drain issues that cause unexpected battery outage as "bugs". In the research work [4], the authors have observed that about three-quarters of abnormal battery drain
issues were caused by application-related bugs and misconfigurations. Currently, most smartphones OS employ power models that describe the behavior of application in response to energy consumption in the device. Findings indicate that incorrectly understanding and implementing these models can lead to excessive use of battery [5]. Furthermore, Internet forums and bugs’ reports give an insight of how severe these bugs are, which caused a great deal of user frustration.

Existing literature [6] has shown that for modern smart phones, the major power consuming components are the screen, CPU, WiFi and GPS. These I/O components can be accessed in the application via a set of system calls (APIs) provided by the Android SDK framework. Moreover, the power management functionality, background services and other hardware sources can only be accessed via a set of system calls [2]. Indeed, most of the energy bugs are uncovered via the invocation of system calls. Basically, most of these energy inefficiencies appear when the application does not access the device resources in an appropriate fashion (e.g. not releasing WiFi/GPS Wake locks or expensive sensor updates), eventually shortening the battery life. For example, Osmdroid, a popular Android application similar to Google Maps, has an energy inefficiency problem that is already reported by users (Osmdroid Issue 53) [7]. The issue was triggered when users switch from MapActivity to other activities without enabling location tracking, which keeps acquiring GPS data to render an invisible map due to an unreleased wake lock. Consequently, battery energy is wasted by the useless location sensing.

In general, bugs’ reports from users are useless and barely provide useful information that the developer can rely on to identify the problem and fix it. Figure 1.1 shows examples of bugs’ reports of battery drain issue on K9Mail. A developer sometimes needs to do an intensive analysis and testing in order to reproduce the issue. Therefore, the best bid is to understand what the user has
K9 and Battery Drain

I use K9 Mail 4-200 on Galaxy note. For weeks it was fine. I use IMAP push feature. Today morning the phone was warmer than expected after charging. After unplugging the charger and not using the phone for an hour I checked the CPU usage using CPU spy and I found the phone did not enter the deep sleep mode. I used the K-9 Push Ridge button to check the time and I found out that the phone is back to normal and was able to enter the deep sleep mode.

Just happened to me today. In less than 6 hours my Nexus 4 used up 82% of the battery. K-9 mail was responsible for 64% of the battery use. Version 4.409 K-9. Up to date stock Android. Have used K-9 for several months. No changes to mail settings. First time this has happened to me.

Since several days I can observe that K-9 consumes so much power that the battery is empty after about 5 hours. Although I have not even touched the phone. In the battery page I can see that K-9 is the root cause. If I stop K-9 in the settings → apps menu, the memory usage is normal (which is easy to see on the graphical curve of the battery).

I see the same problems with the same setup. Started a few days ago. I was getting over 24 hrs battery life, but now can barely reach seven - and that’s with very light use. K9 uses 48% of the power (Android OS 22%, wifi & screen 54% each). Forcing Close doesn’t help - it comes back again even though all accounts are set poll = never. Battery gets warm (38 C).

I’m using K9 Mail for some year now but since I’ve updated to Lollipop >= 5.0.1 (actual 5.1.0) K9 drains my battery. I’m using K9 for four email accounts for over 2 years (or more) and I’ve changed nothing in the last months. But since I’m on Lollipop I’ve noticed that the device (Nexus 5) is sometimes very hot and the battery is drained.

Figure 1.1: Examples of bug reports of battery drain issue on K9Mail

done before the bug being manifested. This motivated us to propose a framework to help developers figure out where and how the abnormal battery drain issue occurred in the app and identify the code regions that need investigation.

In this work, we propose a framework for detecting energy bugs on mobile applications from user interactions. We aim at energy bugs caused by mobile applications, which are potential sources of power consumption. Such software defects usually make the application consume higher amount of energy than expected. Knowing that such energy bugs are exposed via the invocation of system call(s). Therefore, we intend to dynamically detect such bugs on mobile applications from user interactions that potentially invoke such system calls that may lead to unexpected power consumption. Our focus is to identify areas in which the app is consuming power and how this is happening by analyzing inferred user actions and their corresponding power consumption. In order to do that, we use online trackers on the smartphone to obtain the app power consumption and user actions and log them. We then statically analyze the runtime information to identify the power-hungry parts of our app, infer sequences of user actions and report the sequence that triggered the problem. We collect the power consumption data
from the running app using Trepn profiler, a power and performance profiling application for mobile devices [8]. To track the user actions, we instrument the app to track and log app actions triggered by the user while using the app. Both power consumption data and actions logs will then be used to do the final analysis.

We evaluated the effectiveness of our framework in detecting energy bugs from user interactions by applying it to real Android applications. We ran different activities on the concerned application in order to capture user interactions while monitoring for energy bugs. We then analyzed the run-time information to detect energy bugs and identify the corresponding user interactions that triggered them. We inferred sequences of user actions and estimated the power consumption of each sequence by mapping them to the power consumption estimation results. This work aims to help mobile app developers detect energy bugs in mobile applications and identify where and how they occurred in the application. This type of information helps them make their apps more power-efficient. To summarize, we make the following main contributions:

- A framework for detecting energy bugs by monitoring power consumption while running user interactions on Android apps.
- Capturing user interactions with the app to identify the sequence that triggered the energy bug.
- Evaluation of our framework by using it on Android applications with sequences of user interactions.
- Static analysis to identify where and how irregular power consumption occurred.
Chapter 2

Background

This section first presents background information on the energy inefficiencies in mobile applications and how researchers touched this hot topic. It then describes the state-of-the-art in profiling power consumption on mobile devices.

2.1 Energy Inefficiencies In Mobile Applications

A class of bugs, that is energy bugs, identifies energy inefficiencies in smart phone applications. Pathak, et al. [3] have taken a first look at energy bugs on smart phones and focused on the associated challenges in dealing with them. They defined the energy bug as an error in the system- either application, OS, hardware, firmware or external -that causes an unexpectedly high energy consumption. The result is that mobile devices run out of power. Their study has shown that the most prominent type of energy bug is an application-related energy bug. They showed the severity of the abnormal battery drain issues caused by various reasons, including hardware (22.93%), software (35.10%), and other external causes.
Modern smart phone devices are designed to operate at different power levels to make the battery last longer. However, an application with an energy bug may lead to inappropriate power states, such as a non-idle power state in the absence of user activity. Most of these energy problems appear when the application does not access the system resources in an appropriate way, e.g., not releasing Wake locks exported by the Power Manager class in Android [2]. This is primarily due to the flexibility given to developers to explicitly use power control APIs exported by the OS to keep the components on or off during their use by the application. In fact, I/O components are primary sources of energy consumption in smart phones. Current smart phone OSs employ power management policy that every component, including the CPU, stays off or in an idle state, unless the app explicitly instructs the OS to keep it up. Sleep-state transition and background services are common types of application-related energy bugs that can be found in Android applications [2]. Such bugs were observed in many mobile applications, including Facebook, Location listener, email apps [K9mail], dialer app, and GPS based apps [Osmdroid, GPSLogger and Zmanim].

Listing 2.1 shows an example of one possible scenario, background services, which can lead to an energy bug [2]. The application code is supposed to start a location-update background service (Line 10) in the onCreate method. Subsequently, it performs some operation with list data (Line 12). When the user stops the application, the location-update service is removed (Line 19) in the onStopped method. However, if there is an exception before Line 19 (for instance, due to Line 18), the location-update service is never stopped, resulting in an energy bug.

Listing 2.1: Example of code with a potential Energy Bug

// Test Android-Aspectj Project
List<Integer> data;

LocationManager locationManager;

long Min_Update_Time = 10, Min_Distance = 1000 * 60 * 1;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    locationManager = (LocationManager) getSystemService(LOCATION_SERVICE);
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, Min_Update_Time, Min_Distance, this);
    someFunctionToManipulateDataList();
}

@Override
public void onStop() {
    super.onStop();
    try {
        data.clear(); //this can throw an exception
        locationManager.removeUpdates(this);
    } catch (Exception ex) {
        Log.v("Demo", "some exception happened");
    }
}
2.2 Energy Profiling in Mobile Devices

Due to the growing energy demand, research has focused recently on improving the energy efficiency of hardware, and adopting the concept of energy-aware development. This basically depends on the adequate knowledge of how and where energy is consumed on a device. It is essential to have accurate estimations of energy consumption of mobile devices and understand how it has been consumed. Most devices do not provide accurate instantaneous measurements about discharge current and remaining battery capacity because they do not have built-in sensors to get these data. In addition, the energy impact of software can not be neglected. Therefore, different frameworks and tools were proposed for energy metering and profiling.

Pathak et al. in [6, 5] have presented a model based on tracing system calls. Their claim is that hardware component utilization-based models do not address the power behavior of devices, such as camera and GPS, where a process might use the subsystem for a few milliseconds, but in consequence, the operating system keeps that subsystem alive for much longer. They further claim that certain operations on the smart phone have an indirect impact on subsystem power usage. For example, file I/O can cause a change in the power state of the SD card module.

In [9], PowerBooter was proposed to build a battery-level model automatically. The authors proposed using internal battery voltage sensor to detect power consumption, instead of using external metering instrumentation. One approach recommends that developers need a power model that provides accurate, real-time power consumption estimates for power-intensive smartphone components including CPU, LCD, GPS, and audio, as well as Wi-Fi and cellular communication.
components. For instance, in [9], authors have proposed PowerTutor, which implements the PowerBooter model to profile power consumption of applications. PowerTutor allows user to monitor the real-time power consumption of the phone or monitor the power consumption of applications. It displays the total system power consumption decomposed by hardware components including: LCD/OLED, CPU, Wifi, 3G, GPS and Audio.

The work in [10] has presented AppScope, an Android-based energy metering system. This system monitors applications hardware usage at the kernel level and accurately estimates energy consumption. AppScope is implemented as a kernel module and uses an event-driven monitoring method that generates low overhead and provides high accuracy. Another tool, that is related to AppScope, is Trepn Profiler, a hardware sensor-based power profiler by Qualcomm [8]. It is an accurate real time profiler and we use it in this work to get an accurate power measurements.

Existing works such as [2] have shown that I/O components are primary sources of energy consumption in a smart-phone. I/O components are usually accessed in application code via system calls. Moreover, the power management functionality (e.g. Wakelocks), background services and other hardware resources of a device can only be accessed through a set of system calls. In general, most of the energy bugs are exposed via the invocation of system calls. Therefore, to be able to detect energy bugs in such applications, we intend to use user interaction scenarios, which potentially invoke such system calls. This can help detect application-related energy bugs. Furthermore, other research has pointed out that a good approach to detecting energy bugs on mobile applications is to know how such bugs manifest themselves. Lui et al. [11] have found that most Android applications are suffering from performance bugs. They have conducted
an empirical study of 70 performance bugs from real-world Android applications. These bugs can significantly slow down applications or cause them to consume excessive resources such as energy. Another important finding was that most performance bugs need special user interactions to manifest, which means that effective detection approaches has to consider sequences of user interactions with the application.
Chapter 3

Framework Design and Implementation

The goal of our proposed framework is to help mobile app developers figure out whether the app suffers from an energy bug, which causes high power consumption, and pinpoint its location in the code. By having this information, developers can know which part of the app code need to examine, and what actions triggered the issue. In our framework, we prompt such energy bugs to show up by running a sequence of user interactions that are likely to cause irregular power consumption. We capture these interactions and the associated APIs invocations through which the concerned application accesses the hardware components. Examples of such APIs are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Class</th>
<th>API Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Management</td>
<td>PowerManager.WakeLock</td>
<td>WakeLock.acquire()</td>
</tr>
<tr>
<td>Wireless Networks</td>
<td>WifiManager.WifiLock</td>
<td>WifiLock.acquire()</td>
</tr>
<tr>
<td>System Location Services</td>
<td>LocationManager</td>
<td>LocationManager.requestLocationUpdates()</td>
</tr>
<tr>
<td>Telecomm Networks</td>
<td>android.telephony</td>
<td>SmsManager.sendTextMessage()</td>
</tr>
</tbody>
</table>

Basically, our framework aims at two key points: (1) Which part of the code that is suspected to have energy bug, and (2) how the high power consumption was triggered. In the following, we describe our framework, where we first
give an overview of our system and then we detail the design and the implementation of the two key components: the online tracking component, and the offline analysis component.

### 3.1 Framework Overview

Figure 3.1 gives an overview of our framework and its components. The framework is composed of two components: (1) Online-tracking component that collects the logs and the power consumption data from the target device. User actions need to be logged as well as API calls through which the app accesses the hardware components on the device. Each entry will be tagged with a system time for offline correlation; and (2) Offline analyzer that performs offline analysis, and reports sequence of actions and the associated power breakdown. It analyzes the power data and runtime logs and reports the resultant power breakdowns after mapping them to the corresponding sequence of actions. It will then identify a sequence with high power consumption to have energy bug.

To detect energy bugs in the suspect app, first we need to collect both the
power consumption on the phone caused by the app along with the user actions while using the app, and then with these data we can infer sequences of user actions and perform the power correlation. A sequence with high power consumption is reported to trigger an energy bug. The online tracking component on the device collects the runtime information. We use Trepn profiler [8] to provide an accurate power profiling of the app on the device. Using Trepn plug-in for eclipse, we monitor the profiler on the device. After profiling, the plug-in pulls the data from the device and plots it in charts within Eclipse, which we can then analyze. In the same time, the user interactions with the app are logged along with the APIs invocations, which are likely to lead to energy inefficiency. To implement this, we instrument the suspect application to collect the needed information for the analysis component.

When the testing is completed, the PC for offline analysis retrieves the power data and the logs. In the analysis, we interpret the logged actions into sequences and correlate the measured power consumptions to the associated sequence. From the analysis results, we can identify irregular power consumption. Finally, sequences of user interactions that triggered the energy bugs are reported to the developer for further investigation.

3.2 Estimating Power Consumption

We have done extensive research on power consumption models in existing literature. Our concern was to understand the energy consumption of a smartphone applications, and how to estimate energy consumption of applications running on Android smartphones. Android framework provides APIs that can give some indications about the battery level and discharge state during its life cycle.
However, these measurements are not accurate enough to indicate system power consumption. In fact, it is difficult to determine the impact of software design decisions on system energy consumption. A developer needs a power model that provides an accurate, real-time power consumption estimates for power-intensive smart phone components including CPU, LCD, GPS, and Wi-Fi.

Examples of existing power models that may apply in our case are: AppScope [10], PowerTutor [9] and Eprof [6]. However, some of these tools are either not available yet, or support a small range of devices. One related tool that adopts similar methodology of AppScope is Trepn profiler. Trepn Profiler [8], A product of Qualcomm Technologies, Inc., is an on-target power and performance profiling application for mobile devices. It runs on most Android devices, and tablets (Android 4.0 and higher). A wide range of devices have been tested and support battery power measurements, one of them is Google/ASUS Nexus 7, our target device.

3.3 General Background

We implemented our framework on Android to dynamically detect energy bugs from sequences of user interactions using the constructs of Aspect Oriented Programming language (AOP). In order to have a better understanding of our prototype, we give a brief background on some basic concepts used to implement our framework. In the following, we give a general background on Android and AOP in order to help in describing our design.
3.3.1 Android Framework

We implemented our prototype on the Android system. In this work, we focus on Android apps due to their popularity and availability. Additionally, a wide variety of tools are available for Android application developers. This includes tools to monitor the state of an application in real-time (e.g. LogCat) and to communicate with the device (e.g. Android debug bridge). Applications are provided to the user via different application markets like the official Google Play Store and various third-party stores [12]. Most of the applications are written in the Java programming language. Android Apps are distributed as an Android application package (APK). They are compiled to Android’s own byte code format, called the Dalvik executable (dex). On application launch, the Android middleware spawns a new Dalvik Virtual Machine to execute the application’s dex file. This enables Android to exploit the process isolation mechanisms of the underlying Linux operating system and ensures that all applications are run inside their own isolated containers [13].

An Android application is a single installable unit which can be started and used independently of other Android applications. An Android application can have one application class which is instantiated as soon as the application starts and it is the last component which is stopped during application shutdown. An Android application consists of Android software components and resource files. Android application components can connect to components of other Android applications based on a task description (Intent). This way they can create cross-application tasks. The integration of these components can be done in a way in which the Android application can still work flawlessly, even if the additional components are not installed or if different components perform the same task.
Android applications do not have a single entry-point, such as the main method in Java; instead, the application is designed in terms of components. Every component is implemented as a Java class derived from a specific base class in the Android middleware. Components react to OS events by overriding the respective methods or calling specific OS methods to register further callbacks that are invoked when e.g. device's physical location changes [13].

There exist four components that can be defined in Android applications: activities, services, content providers, and broadcast receivers [14]. Activities are single focused activities a user can interact with. They are the visible parts of an application. In contrast, the services run in the background and are not interacting with the user directly. They are used for long-running background operations, such as MP3 playback and GPS service. Broadcast receivers react to global events, such as incoming calls or text messages. They define how an application responds to system-wide messages. Content providers manage shared application data, and provide an interface for other components or applications to query or modify these data for, e.g. Contacts [14]. Each of the four different types of components has a distinct life cycle that defines how the component is created, used and destroyed. The life cycle is guided using events, i.e., a sequence of methods called by the OS. Each component starts with a call to the onCreate() handler, and ends with a call to the onDestroy() handler [14]. Figure 3.2 shows an activity life-cycle [7].

3.3.2 Aspect Oriented Programming

We use the language constructs of AOP to help implement our approach. As AOP is a relatively new software development approach, researchers tended to focus on design and development, while software testing had less attention [15].
Since its introduction by Kiczales in [16], AOP has been an asset in many software development projects, a supporting technology for many research areas, and a research subject on its own. It has been used in several techniques for software reliability and verification such as run-time verification and testing [17].

The main purpose of AOP is the separation of cross cutting concerns [16]. When we talk about cross cutting concerns we are referring to generic functionality that is used in several places in our system or application. These concepts are logging, transaction management, error handling, monitoring and security. When using AOP for performance monitoring, the code required to perform the measuring can be easily added or removed due to the weaving process used by AOP [18]. Using AOP with Android application is a good approach because an Android application is developed using a well identified set of methods defined in the SDK. So it is easy to write an aspect implementing abstract functionality that works with any Android application. Many compilers are available to implement AOP. One example is AspectJ that can be used to instrument Android applications.

Aspects rely on three concepts: JoinPoints, Pointcut, and Advice. A JoinPoint is an identifiable point in the execution of a target program. We execute some code
(advice) each time a join point is reached; when a method is called for example. A pointcut selects a set of joinpoints. Advice defines crosscutting behavior. It is a piece of code associated to some pointcuts. The code of a piece of advice runs at every join point picked out by its pointcut. AspectJ supports three kinds of advice, which determines how it interacts with the join points it is defined over. An advise can run before its join points, after its join points, and in place of (around) its join points.

Figure 3.3 shows an example of an aspect that were woven into an Android application [17]. The aspect is intended to be woven into the Android game application, Tic-Tac-Toe. The aspect determines whether Tic-Tac-Toe invokes a Google banner by catching any call to the methods in the com.google.ads package. When such a call occurs, the aspect blocks the banner using the around advice that cancels the call.

### 3.4 Online-Tracking Component

The online tracker on the smartphone is used to collect the runtime information. It mainly tracks two things, app power consumption and user interactions with the app.
We track the app power consumption using Trepn profiler. It profiles in real time how a mobile app uses CPU, network and hardware resources. We use it along with the Trepn plug-in for eclipse. It is a tool for Android app developers, designed to allow them to easily collect performance data from any app running on a mobile device, and analyze the resulting graphs in the Eclipse IDE. We have decided to use the Trepn package among other existing tools such as, Eprof [6] and AppScope [10], as it supports our device, ASUS Nexus 7, besides a broad range of commercial Android devices. In addition, the plugin-in provides more flexibility to the developer as it is fully integrated with Eclipse, the development environment we use to implement our prototype. With Trepn plug-in, we can view and manipulate all of Trepn’s power metrics and statistics easily from Eclipse. We also do not need to use extra hardware and any external metering device. Figure 3.4 shows an example from K9Mail app we profiled on a Nexus 7 tablet.

When we connect our target device to the PC, the plug-in will push the Trepn APK to the device to profile and send data back to Eclipse for off-device analysis. Before starting, we set up the profiler to pick which of the system resources we acquire its data. Trepn plug-in for Eclipse gives access to data including: Power at battery, CPU frequency and utilization, Display and GPU, and others. However, since we are concerned about the power consumption, we pick the power usage at the battery that represents the amount of power consumed by the app. Trepn charts output at the battery in watts at 100ms intervals. We then correlate the profiling data to the entries in the LogCat using the timestamps.
Figure 3.4: Using Trepn to profile battery power relative to CPU on K9Mail

3.5 Actions Tracker

The main goal of our framework is to find out where the battery drain problem of an app occurred and what triggered it. Therefore, we capture the user interactions with the app in order to identify the trigger of the energy bug; what causes the bug to manifest itself. Sometimes, the user cannot remember or figure out what he has done before the problem occurs. Indeed, we track app actions and record what a user has done so that the developer can indicate what has triggered the issue. Additionally, Given that most of the energy bugs are exposed via the invocation of system calls, we capture these calls to investigate why the
problem occurs. Therefore, to be able to trace these calls in such applications and
detect abnormal power consumption they may cause, we apply user interactions
scenarios, which potentially invoke such system calls.

We implement this component by instrumenting the application under
test. We capture the user interactions with the app, and log them. Within these
actions, we capture the calls of any power control APIs as well. In our instrumen-
tation, we target those methods related to user interaction event. Table 3.2 lists
the two main categories of Android APIs we target. The resulted logs will then
be sent to the offline analysis component. After the analyzer receives the logs, it
will estimate the power consumption of each sequence of user actions by mapping
them to the measured power consumption.

The first category is Android Activity life cycle related methods [19]. The
user interaction interface of an Android application is referred to as an Activity.
An activity can be in one of the seven stages during its lifecycle. Usually, all
the set-up tasks (such as acquiring resources and starting background services)
take place in four stages of the activity, namely onCreate, onStart, onResume
and onRestart. Similarly, the entire tear down tasks (such as releasing resources
and stopping background services) take place in three stages, namely onPause,
onStop and onDestroy. The second category is user interface (UI) related input
events [20], which indicate the interaction between users and UI components.
These are related to everything shown on the screen that the user can interact
with.

In order to instrument the concerned application, we use the constructs
of Aspect Oriented Programming language. We develop an aspect that captures
the runtime information and log them in the Android’s logging system for later
Table 3.2: Android APIs related to user interactions sequences

<table>
<thead>
<tr>
<th>Category</th>
<th>APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity Life Cycle Related</td>
<td>onCreate, onStart, onRestart, onResume, onPause, onStop, onDestroy, onSaveInstanceState, onRestoreInstanceState</td>
</tr>
<tr>
<td>UI Related</td>
<td>onClick, onLongClick, onFocusChange, onKeyDown, onTouch, onDrag, onCreateContextMenu, onContextItemSelected, onContextMenuClosed, onCreateOptionsMenu, onOptionsItemSelected, onOptionsMenuClosed, onMenuItemClick, onListItemClick, onItemClick, onItemLongClick, onItemSelected, onNothingSelected, onChildClick, onGroupClick, onGroupCollapse, onGroupExpand, onDoubleTap, onDoubleTapEvent, onSingleTapConfirmed, onLongPress, onScroll, onScrollStateChanged</td>
</tr>
</tbody>
</table>

analysis. It logs the invocations of system calls as well as the actions that triggered them.

3.6 Offline Analysis

We then analyze the run-time information to identify where and how irregular power consumption occurred, if it exists. In our analysis, we use Android tools such as DDMS (Dalvik Debug Monitor Server) and Traceview [21]. These tools aid in reading, filtering and analyzing the Android log system. After analyzing the resultant logs, we infer the app actions and interpret them into sequences of user actions. To recall, we track app actions by recording associated invoked methods. To infer a sequence of actions, we look for the method calls of each action from the log. Next, we estimate the power consumption of each sequence by mapping them to the power consumption estimation results from Trepn. Using the Trepn plug-in, we plot charts to help analyzing the power consumption measurements. We calculate the power consumption of each sequence by summing up
the energy consumption of individual actions during the course of each sequence. The system time is used to map the power data to the individual actions.

In order to get the system time, we used Android Traceview [21]. We used Traceview to record the time when the method starts to run and its return time. The following shows the algorithm that we use to associate power data with the corresponding method and then estimate the power consumption.

1. Assume the log period is $T$

2. The time when a method starts is $t_1$

3. The time when a method returns is $t_2$

4. The average power consumption of a method (action) is $P_A$, and calculated as the power changes over the life time of the method:

$$P_A = (P(t_2) - P(t_1))$$  \hspace{1cm} (3.1)

5. The power consumption of a sequence is $P_S$, and calculated as the sum of the power consumption of individual actions (methods) involved:

$$P_S = \sum (P(A_1) + P(A_2) + P(A_{i+1}) + ... + P(A_n))$$  \hspace{1cm} (3.2)

According to the power consumption estimation results, a sequence with high power consumption would be reported as suspected to have an energy bug. At that time, a developer can refer to the log to identify the API calls that might be responsible for such a bug.
Chapter 4

Evaluation

We evaluated our framework to answer the following research questions:

RQ1 Is our framework able to dynamically detect energy bugs in a suspect application?

RQ2 Is our framework able to identify how the energy bug was triggered?

RQ3 Is our framework able to detect other energy bugs that can occur simultaneously due to other factors in the system (other applications or OS)?

First, we describe our experimental setup and the set of subject programs that we analyzed in our experiments.

4.1 Experimental Setup

We performed our experiments on a Nexus 7 tablet running Android 5.0.1. The device uses standard I/O components such as GPS and WiFi. We evaluated our approach by using two test cases Android applications, K9mail and Osmdroid. We downloaded the source code for the concerned applications
from their Github repositories. We instrumented the applications by injecting the Aspect that captures and logs the user interactions and the API calls. We used AspectJ compiler to integrate the aspect into the Android projects. We used Eclipse plugin for AspectJ Development Tool ADT bundle. First, we configured each Android project to be AspectJ aware. Next, we exported the AspectJ runtime library so that the Android toolchain picks it up and compiles it into the apk. The power consumption data was acquired through the Trepn profiler that runs on the device. We used Trepn plug-in to control Trepn from the computer and allow off-device analysis. Trepn plug-in is integrated in our Eclipse IDE. It helps to easily collect data from apps running on the mobile device and analyze the resulting graphs in the Eclipse IDE. The diagram in Figure 4.1 describes the experiment setup process.

4.2 Integrating AspectJ Within Android

Although programming applications for Android is mostly in Java, it doesn’t run Java byte-code. Android uses the Dalvik VM, as opposed to the Java VM. Understanding the build process of Android applications helps to know how aspects are compiled and weaved to an application. In the weaving process, the code (advise) is injected into the target places (jointpoint). We use source-code weaving, where we add an extra step to the build process to modify the compiled classes before packaging and deploying the application. Once the Java compiler compiles the java to Java byte-code (.class) files, AspectJ compiler injects point cuts and advices to the .class files. Then Dex takes these new class files and creates Dalvik bytecode (.dex) file.
When compiling an AspectJ project, it first compiles classes and aspects to Java byte-code. Then an AspectJ Weaver weaves the Aspect byte-code into the Class byte-code. Within the build process we can see that the class files are changed to Dalvik bytecode using the Dex tool. This means that we need to add a process after the javac and before the dx tool, instead of the use of the AspectJ builder, which compiles and weaves. This means that weaving can only be done at compile-time and not at run-time, because dynamic weaving would only create Java bytecode, not Dalvik.
4.3 Aspect Code

We have developed our aspect that is responsible for capturing user interactions and system calls. The aspect logs all Activity lifecycle and user interactions related methods that can be retrieved using LogCat. Our target methods fall in the list of APIs mentioned before in Table 3.2. The Aspect mainly consists of two Pointcuts and one Advice. While the Advice defines what the cross-cutting functionality is (logging), the two Pointcuts define where it should be weaved in (whenever our target methods are called). Listing 4.1 shows the two pointcuts defined in the developed aspect.

Listing 4.1: The Aspect that captures user actions with the app

```java
package ...android...Aspects;

import android.util.Log;

public aspect ActionsTrackerAspect {
  ...
  pointcut ActivityMethods() : execution(* Activity+.on*(..)) ;
  pointcut UIMethods() : execution(* View+.on*(..)) ;

  // Advice that gets executed "around" the Pointcuts ...
  ...
}
```
4.4 Subject Applications

For the sake of our evaluation, we used our framework on two open source Android applications selected based on the following criteria:

- We consider applications with common energy bugs to minimize false positives.
- We consider applications that are user interactions driven so we can run sequences of user interactions that are more likely to uncover potential energy bugs.
- They represent different categories of applications.
- They provide two different services: the sleep-state transition and background services, which are two common types of application-related energy bugs that can be found in Android applications.

In the following, we will introduce our two test cases, K9mail and Osmo-
droid.

1. K9Mail [22] is a popular open-source Email client on the Android platform, and also we used it as an experiment subject of our framework. We used a bug in K9Mail, which has been reported by many users. The bug is triggered by incorrect settings of IMAP connection to the remote server. Figure 4.2, shows the bug report as a battery drain issue on the K9Mail website. It is clear that the information provided by the user does not give any useful information to the developers, which requires an intensive analysis and testing to figure out why and how the problem has occurred. Therefore tracking the actions that triggered the issue would help to locate the problem, and this is the key idea of our framework.
2. OSmdroid [7] is a navigation application similar to Google Maps. The bug reports indicated that there was an energy bug in this app, which represents background services type of energy bugs. Figure 4.3 shows the code that has the potential energy bug in OSmdroid. Basically, when MapActivity is launched, it starts GPSService (Lines 5-6), and registers the broadcast receiver (Lines 15-16). GPSService registers a location listener with the Android system when it starts (Lines 36-47). When the user’s location changes, GPSService would 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 the map (Line 10). If the user has enabled location tracking, these data would also be stored in a database (Line 11). If the Android system 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) [7]. If OSmdroid’s users switch their smart phones to another application, MapActivity would be put to the background (not destroyed), but GPSService would still keep running for location sensing. If the location tracking functionality is not enabled, all location data would be used to refresh an invisible map. Then, a huge amount of energy would be wasted. To fix this problem, developers later disabled GPS sensing if users leave MapActivity without the location tracking functionality enabled.
4.5 Experiment Design

The main focus of our experiment is to observe the ability of our framework to uncover energy bugs by running a sequence of user interactions on real world applications and identify what has triggered them. To detect energy bugs using our framework, we ran different scenarios of actions on our test subjects including the suspected scenario that has already been reported by developers. To get the needed data for our experiment, we have run the instrumented apps, each for 5 minutes with Trepn profiler running in the background. In order to obtain relevant data, we have tried different scenarios that are maintaining different states of utilization for the hardware components on the device. To avoid bias, we have stopped running application and anything we didn’t want considered in our power readings including Android housekeeping and apps updates.

In each session, we have pushed the instrumented version of the app to the target device. In the background, Trepn profiler was running to profile the running app and acquire the power consumption. We controlled the profiler using the Trepn plug-in from the eclipse IDE on the computer. The plug-in pushed the APK (Trepn profiler) to the USB-connected device and then we started Trepn Profiler on the device after picking the app to profile. Meanwhile, the aspect
logged user interactions with the app along with the invocations of APIs. In
Figures 4.4 and 4.5, we show how interactions with K9Mail app were captured
by our aspect. At the end of each session, the plug-in pulled the data from the device
to start the off-device analysis. To perform our analysis, we used Android tools
such as DDMS (Dalvik Debug Monitor Server) and Traceview [21].

By analyzing the log, first, we inferred the user actions and interpreted
them into sequences. To do that, we looked in the log for the pair of the method
entrance and exit of each action. The sequence could be composed of one method
call or nested method calls. Next, using the Trepn results, we calculated the power
consumption of each sequence based on the above mentioned algorithm. Actually,
we mapped each sequence of actions with the measured power consumption based
on their occurrences in the LogCat. Based on the analysis results, we generated
our final report that would guide developers in solving potential energy issues.
4.6 Results and Data Analysis

One of the objectives of our experiments was to observe the ability of our framework to uncover energy bugs in real-world applications by running a sequence of user interactions, and identify how the energy bug was triggered?

Our first experiment was with Osmdroid. We have investigated the reported bug and observed that there is a sequence of user interactions that triggers the bug. The energy bug reported in this app represents background services type of energy bugs. It happens due to holding wakelock for hours, along with radio communication of GPS and network. The bug is triggered when users switch from MapActivity to other activities without enabling location tracking; location data would then be used to render an invisible map. Profiling Osmdroid with different actions has resulted in the graph in Figure 4.6.
To verify a reported bug using our framework, we have run the version of the app with the energy bug using our framework. We have executed the sequence of actions: starting the app- select the sample loader- select the first sample and suspended the application afterwards by pressing the Home button. At this point, location sensing is not needed anymore and must be disabled. We have captured the power consumption and user’s actions. Figure 4.7 shows the measured power consumption of the application. Per the developers, the application had a missing code fragment for disabling the location updates when exiting the app. Adding the release code at the correct location has fixed the issue. Therefore, we re-ran the fixed version of the app with the same scenario of actions using our framework. Figure 4.8 shows the power consumption of the fixed version of the app.
the two graphs, we can see that the app has consumed more power in the first run than the second run (after adding the release code). Then, we have applied our algorithm to estimate the average power consumption of the selected actions. Figure 4.9 shows the average power consumption per action. The blue line indicates the actions that were executed with the impact of the energy bug, and their power consumption readings were higher than those without being affected by an energy bug as indicated by the green line. Indeed, the actions in the first run (before the fix) have more average power consumption than those in the second run (after the fix). Therefore, our framework has succeeded in reporting the sequence of actions with energy bug in Osmdroid.

The second subject was K9mail. To verify our proposed framework, We
have applied sequences of actions such as , navigating through the Message List, composing a new Email and sending it out and navigating back to the Message List. We then used a bug in K9Mail [22], which is triggered by incorrect settings of IMAP connection to the remote server and has been reported by many users. Some reports recognized that there were too many IMAP connections, while most reports only knew that the CPU utilization was high and it drained the battery. This bug is caused by the number of IMAP connections set in the app exceeding the connection limit allowed at the remote server, and it can be manifested if user manually increased the number of the "push" folder over the limit. For example, Gmail now has a limit of 15 simultaneous IMAP connections per account, and by default one connection is for one folder synchronization activity. Therefore, We have changed the account push settings to increase the connection numbers to have more folders synchronized at a time, and after awhile we have noticed slow

Figure 4.9: Average power consumption of the Actions in the two runs of Osmdroid
After analyzing the power data and the logs, we have applied our algorithm to perform the mapping and estimation. By using the action tracker, we obtained sequences of actions, and the corresponding average power consumption as shown in Figure 4.10. Figures 4.11 and 4.12 show the actions inferred from the K9Mail Log shown above. During one run of K9Mail, the user browsed the messages’ list, composed a new email when being in the inbox and sent it out.

We have noticed that some of the sequences instances have much power consumption than the other. However, there was no impact of an energy bug in our application. The reason behind that might be the absence of defects in the App’s version that we have used in our experiment.
On the other hand, our framework has detected irregular energy consumption that can be attributed to energy bugs in other applications or other external factors in the system. By profiling the system, in a desire to observe the ability of our framework to detect other energy bugs, we were able to uncover other types of energy bugs that were related to external conditions such as wireless signal strength and failure in Android Adapter service synchronization. Profiling the system has shown power spikes and jumps in power consumption as in Figure 4.13.

After correlating power spikes to the corresponding events in LogCat, two System Server processes were observed to cause high power consumption. First, we have observed that the high power consumption was due to weak wireless signal
strength that caused NIC drivers to compensate by increasing its power. This has significantly increased the battery drain of apps that perform network activities, e.g., background processes that perform periodic polling. Figure 4.14 shows the corresponding entry captured in the LogCat.

Next, as shown in Figure 4.15, we have observed another example of energy bugs that occurs when the Adapter Service fails to Synchronize. Such conditions trigger the clients repeatedly trying to connect to the remote server, perform email authentication, or ping the server draining battery on the phone. This error showed up several times in the system log at different runs, and it was due to an email application installed on the device. Furthermore, after researching the problem, we have found that the same error was reported in the Android open Source Project Issue Tracker as "Repeated Email Sync Failure Eats CPU and Battery ". Basically, the android device fails to sync due to authentication error,
and never stops trying throughout the day, causing high CPU usage and battery drain.

4.7 Analysis Report

Following the analysis, our framework generates a report that contains sequences of actions and the associated power consumption. This report serves as a guide to optimize energy consumption and investigate source code for potential energy bugs and remove them. In addition, the report may help the developer to prioritize bugs and energy issues based on the power consumption results. For
Figure 4.14: The LogCat is showing an energy bug due to OS processes (weak wireless signal)

Figure 4.15: The LogCat is showing an energy bug due to OS processes (Failure to Adapter service synchronization)

each case, the developer may run the reported sequences of actions and observe the sequence that crosses the application to trigger the energy issue. This would aid in identifying the root cause of the energy bug, and system calls that may be responsible for such an issue.
Chapter 5

Related Work

We proposed a framework to help detect energy bugs from user interactions. First, to gain a better understanding of energy bugs, how they manifest, and what are the root causes, we have studied related existing research. In [3, 23], the authors defined energy bugs and identified their root causes. They proposed a road map towards developing systematic frameworks treating these energy bugs. In additions, Based on the research directions and recommendations suggested in [11], we present a framework for detecting energy bugs on Android apps considering sequences of user interactions that manifest such bugs. In this work, we profile the power consumption of the application while running sequences of user interactions; we capture these interactions using aspects that we weave in the concerned application. We analyze the run-time behavior to verify the manifestation of such bugs.

Our work in this paper relates to different research lines of existing work related on detecting energy bugs, energy profiling on mobile devices, and software testing and profiling using AOP.
5.1 Energy Profiling on Mobile Devices

Several energy profiling schemes have been proposed for mobile devices. Recent research has developed different power models to estimate a device’s power consumption. In [24], the authors proposed Sesame, which is an automatic smartphone power modeling scheme using a built-in current sensor. Their work focused on overall system power rather than individual hardware components. However, this feature does not apply at the application level. Therefore, running time profiling has been proposed at the application level to monitor the call graph trace and estimate the running time of routines as in [5]. Powerscope [25] measures power using an external power meter and accounts energy for mobile systems at the routine granularity. Pathak et al. [6] proposed a system-call-based power modeling to achieve fine-grained online energy estimation for both utilization-based and non-utilization-based power behavior.

Pathak et al. [5] proposed Eprof, the first fine-grained energy profiler for smartphone apps. Eprof is capable of measuring intra-app energy consumption and gives insights into energy breakdown per thread and per routine of the app. It adopts the last-trigger accounting policy to most intuitively capture asynchronous power behavior of modern smartphone components in mapping energy activities to the responsible program entities.

The recently proposed Cinder [26] and PowerTutor [9] also perform smartphone energy accounting. They differ from Eprof in several aspects as they support processes as the finest accounting granularity. Both systems use utilization-based power models to model energy of each component to the processes. Yoon et al. [10] presented AppScope, an event-driven energy metering system, which monitors an app’s hardware usage at the kernel level and estimates the energy consumption.
AppScope was developed as a kernel module. Another related tool is Trepn Profiler, a hardware sensor-based power profiler by Qualcomm [8]. It is an accurate real time profiler and we use in this work to get an accurate power measurements.

Mittal [27] presented an energy emulation tool that helps developers to estimate the energy use for their apps during development. Shye et al. [28] characterized the mobile phone power consumption by user activities, and proposed a regression-based estimation model, which not only estimates the power consumption, but also provides the power breakdown among hardware components. In addition, they studied user behavior patterns to derive power optimization by reducing the screen brightness over time. Zhang et al. [9] used the built-in battery voltage sensor and the battery discharge behavior to build a power model, PowerBooter, which requires no external measurement devices. They also developed PowerTutor, an online power estimation tool, by using the model built by PowerBooter. Balasubramanian et al. [29] presented a study of the energy consumption characteristics of mobile network interfaces (3G, GSM, and Wi-Fi), and developed a power model for each of the interface. Also they proposed a protocol, TailEnder, to reduce energy consumption of mobile apps.

5.2 Detecting Energy Bugs

There have been initial attempts focusing on smart-phone application performance. For example, Pathak et al. studied no-sleep energy bugs in Android applications and used reaching-definition data-flow analysis to detect such bugs (e.g., an application forgets to unregistered a used sensor) [30]. Zhang et al. presented an automated detector of energy leaks for smart-phone applications [31].
In [7], the authors proposed a cost-benefit analysis to detect energy leaks caused by improper or ineffective uses of smart-phone sensors. They actually proposed a run-time analysis technique to automatically analyze sensory data utilization at different states of an Android application in order to identify those application states where sensory data are underutilized. Their tool implementation ranks sensory data utilization coefficients at different states so that energy inefficiency reports can be prioritized. In addition, their tool provides detailed event handler calling traces to help developers construct concrete test cases to reproduce specific sensory data utilization scenarios. Indeed, their goal is to help locate energy inefficiency problems in Android applications.

Another work is an automatic detector of energy leaks for smart-phone applications [31]. The authors focused on detecting and isolating energy leak in network communication including Wi-Fi and 3G interface. In later work [11], the same authors conducted an empirical study of 70 performance bugs from real-world Android applications. Their study revealed some unique features of performance bugs in smart-phone applications. They also identified some common bug patterns, which can support related research on bug detection and performance testing.

In [7], the authors presented an approach for diagnosing energy inefficiency problems in Android applications. Their approach simulates the runtime behavior of an application, and automatically analyzes its sensory data utilization. In another work [32], the authors presented a framework for energy bug diagnosis. Their framework diagnoses the suspect app to see whether it suffers from the abnormal battery drain, and also tries to find out where and how it happened by analyzing energy consumption of app actions. In this work, we present a framework for detecting energy bugs dynamically by tracing user interactions with the app in order to aid in optimization energy consumption on mobile applications.
5.3 Performance Testing

Performance testing is challenging and some efforts have been done to overcome this challenge. For example, Jiang et al. used performance baselines extracted from historical test runs as tentative oracles for new test runs [33]. Grechanik et al. learned rules from existing test runs, e.g., what inputs have led to intensive computations. They used such rules to select new test inputs to expose performance issues [34]. These ideas work well for PC applications, but it is unclear whether they are effective for smart-phone applications. In [11], the empirical study discloses that many performance bugs in smart-phone applications need certain user interaction sequences to manifest. Such requirements and features should be considered in order to design effective techniques to test the performance of smart-phone applications. There have been some initial attempts in this direction. Yang et al. tried to crash an Android application by adding a long delay after each heavy API call to test GUI lagging issues [35]. Jensen et al. studied how to generate user interaction sequences to reach certain targets in an Android application [36]. Our work may be considered as an extra effort along the same direction.

5.4 Debugging and Optimization for smart phone applications performance

Several researches estimates performance for smart phone applications to aid debugging and optimization tasks [37, 38, 6, 9]. Another category of existing work uses profiling to log performance related information to aid debugging and
optimization task [39, 40, 41]. Recently, [42] have presented ProfileDroid, a monitoring and profiling system for characterizing Android app behaviors at multiple layers: static, user, OS and network. Their tool uncovered inconsistencies and surprising behaviors. The work was more concerned with privacy and security issues.

5.5 Testing using Aspect-Oriented Programming

In [43], the author have presented and demonstrated significant potential utility in an approach to using aspect languages to express test adequacy criteria relative to crosscutting concerns. Their approach allows tester intentions to be represented abstractly within source code. Yingzhe et al. [44] presented a new user-friendly Hook-based testing approach for embedded software. Besides introducing the concept of Hook-based testing, they showed how to use AOP to implement components testing on the target easily.
Chapter 6

Conclusion and Future Work

In this work, we proposed a framework for detecting energy bugs dynamically on mobile applications. We targeted energy bugs caused by mobile applications, which are essential sources of power consumption in mobile applications. Knowing that such energy bugs are exposed via the invocation of system call(s), we used user interactions scenarios that potentially invoke such system calls leading to unexpected power consumption. Using our framework, we were able to uncover energy bugs in real-world applications. Some of those bugs were reported previously by users and developers. We have applied different activities on the concerned applications in order to capture user interactions while monitoring power consumption in a desire to detect energy bugs and identify the trigger of the bug. Our analysis resulted in a report that contains sequences of actions and the associated power consumption. It acts as a guide to optimize energy consumption and fixing energy bugs. In addition, the report would help the developer to prioritize energy problems. Additionally, our framework detected other energy bugs that were initiated due to other applications trying to perform network activities, e.g., background processes that perform periodic polling and authentication verification tasks. This work aims to help mobile app developers detect energy
bugs in mobile applications and locate the regions in the code that need inspection and optimization.

Our next step is to validate our framework with more subjects to assess its effectiveness in detecting energy problems. Besides, we would like to investigate energy problems that occur due to external factors such as the OS. We will use our framework to detect new energy bugs. In this work, we did not specify a coverage criterion for energy bugs detection. In the future, we plan to determine a coverage criterion that is considered efficient to help uncover most of the energy bugs in an application. Moreover, a good approach to figure out users’ interactions that may lead to potential energy bugs on mobile applications is to understand how users interact with and move through the app. Therefore, we plan to conduct a study on real applications’ users to investigate their behaviors that lead to the manifestation of energy bugs. Furthermore, with more investigation to the runtime information, we may identify root causes of energy bugs and give guidelines to the developers in order to help testing their applications and make sure they are free of potential energy bugs.
Bibliography


