Mobile Energy Bug Diagnosis

A Thesis

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

By

Jau-Yuan Chen,

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2013

Master’s Examination Committee:

Feng Qin, Advisor

Xiaorui Wang
© Copyright by
Jau-Yuan Chen
2013
Abstract

Smartphones have become the most prevalent computing platforms because of their portability and powerful hardware equipped. However, the battery development barely catches up with the need for higher energy consumption requirement. The worse is software defects or careless developers usually make the application consume exceptionally higher amount of energy than expect and drain the battery in short time.

In this work, we aimed at the abnormal battery drain issues, namely energy bugs, caused by mobile applications and proposed a diagnosis framework to address the issue. We used the online application and system utilization trackers in conjunction with unsupervised clustering and labeling to find out whether the suspect app was suffered from energy bugs, and also we interprets application actions into action units to provide the high level context information for better understanding to both general users and developers. Our experimental results show that our framework can provide high precision and low false alarm rates in diagnosis. We also provides a potential extension to further improve our system.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. System Design and Implementation</td>
<td>4</td>
</tr>
<tr>
<td>2.1 System Overview</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Part I: Online Tracking</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Utilization Tracker</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Action Tracker</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Part II: Offline Diagnosis</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1 Per-app Power Consumption Estimation</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2 Action Unit Inference</td>
<td>20</td>
</tr>
<tr>
<td>2.3.3 Unsupervised Diagnosis</td>
<td>20</td>
</tr>
<tr>
<td>3. Evaluation</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Experimental Setup</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Test Case: K9Mail</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Power Modeling Results</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Energy Consumption of Action Units</td>
<td>27</td>
</tr>
<tr>
<td>3.5 Diagnosis Results</td>
<td>29</td>
</tr>
<tr>
<td>4. Extension: Diagnosis with Bayesian Learning</td>
<td>34</td>
</tr>
</tbody>
</table>
5. Related Work ................................................................. 36
   5.1 Power Modeling and Estimation for Mobile Devices .......... 36
   5.2 Mobile Energy Bug Diagnosis ................................. 37

6. Conclusion ............................................................... 39

Bibliography ............................................................... 40
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The utilization record generated by the utilization tracker every 500 milliseconds.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Android APIs monitored by the action tracker.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Power model for components, the Wi-Fi base, the phone base and the whole phone energy consumption (for building the power model). The training session is divided into ( k ) 10-second sessions (( \Delta(t_k, t_{k+1}) ) is around 10 seconds) to get the per-session average workload of each component and the base energy consumption, and then the average workload is applied to each tracker record that resides in the ( k )-th session for the modeling. ( \Delta(t_{(k,i)}, t_{(k,i+1)}) ) is around 500 milliseconds.</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Action units inferred from Listing 2.1.</td>
<td>19</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Energy bug diagnosis system architecture.</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>A bug report of battery drain issue on K9Mail, which mainly describes the symptoms with little helpful information for the developers.</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Android APK building process and the instrumentation.</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Measured energy consumption and component utilization when training CPU.</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Measured energy consumption and component utilization when training display.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Measured energy consumption and component utilization when training the wireless transmission.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Power model accuracy (estimated results compared with measured results).</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Average energy consumption of action units.</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Accuracy of clustering results.</td>
<td>29</td>
</tr>
<tr>
<td>3.7</td>
<td>Recall of clustering results.</td>
<td>30</td>
</tr>
<tr>
<td>3.8</td>
<td>False alarm rate of clustering results.</td>
<td>31</td>
</tr>
<tr>
<td>3.9</td>
<td>Precision of clustering results.</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Potential extension of the energy bug diagnosis framework.</td>
<td>35</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Smartphones were shown to overtake the personal computer (PC) market since 2011 [1], which had 62.7% sales growth in 2011, and have become the most prevalent personal computing platforms. In the most recent Gartner report [2], their estimate also showed that the mobile devices will continue to dominate the market, and in 2012 the mobile phone sales reached 5 times as the amount of PC. In addition, smartphones today are essentially ultra-portable computers equipped with powerful hardware (e.g. a multi-core CPU and a 3D graphics chip). However, the battery development is much slower than the hardware advances [3], and therefore, the battery life has become the most critical resource in smartphones.

In 2011 when Apple [4] just released their latest iPhone and iOS, which are one of the most popular smartphones and mobile operating systems respectively, they admitted an issue of the battery life on the phone [5], and eventually they found out that it was a software bug in the iOS and took several weeks to fix it. In the recent research work by Pathak et al. [6], they showed the severity of the abnormal battery drain issues cause by various reasons, including hardware (22.93%), software (35.10%), and other external causes. They refer to such abnormal battery drain issues that causes unexpected fast battery depletion as “energy bugs”. In the research work by Ma et al. [7], about three-quarters of abnormal battery drain issues observed
by them were caused by application-related bugs and mis-configurations, and only 7.6% were normal but heavy battery use. This observation also gives us an insight that usually normal battery use does not tend to incur exceptionally high energy consumption on smartphones.

In this work, we present a framework for energy bug diagnosis. We aimed at the energy bugs resulted from mobile applications (apps), which occupy more than one third of software energy bugs [6]. Instead of finding the root cause of an energy bug, our 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 inferred app actions and their corresponding energy consumption. To obtain the necessary information, we developed two trackers on the smartphone that can log app actions triggered by the user and utilization of system components, and then according to the runtime logs, the diagnosis agent at the remote server will output the diagnosis results. We built our prototype on the Android system, and we also developed a configurable and automatic instrument tool for developers, so they can release a version of their apps that support the tracker. Moreover, we did not modify the operating system nor used anything specific to the Android, so our framework is portable to other mobile platforms as well.

The rest of the thesis is organized as follows. In chapter 2, we describe our diagnosis framework, where we first give an overview of our system and then we detail the design and the implementation of the two major components: the trackers at the smartphone side, and the diagnosis agent at the remote side. In chapter 3, we demonstrate the evaluation results of our prototype and present an extension design
in chapter 4. At the end, we summarize related work in chapter 5 and conclude our work in chapter 6.
Chapter 2: System Design and Implementation

The goal of our proposed framework is try to help both general users and mobile app developers figure out whether the suspect app suffers from the abnormal battery drain issues and where and how the issue occurred in the app. By knowing the information, the suspect code region that developers need to exam might be effectively reduced, and users could bypass the battery drain issue temporarily before the fix being released by not doing those suspect actions. Therefore, our framework does not focus on locating the root cause of the abnormal battery drain, but would be an auxiliary tool for developers and user to figure that out. To be more specific, our framework aims at two key points: (1) from which step the app started going wrong, and (2) how the abnormal battery drain was triggered.

In the following, we first give an overview of our proposed energy bug diagnosis framework, and then we describe each key component in the framework and our implementation of the prototype.

2.1 System Overview

Figure 2.1 shows the design of our energy bug diagnosis framework and the workflow. The framework contains two parts: (1) online tracking on the smartphone,
which includes the utilization tracker and the app action tracker; and (2) offline diagnosis on the remote server, which consists of the app power consumption estimation module and the energy bug analyzer.

To apply the diagnosis to a suspect app, first we need to collect both the component utilization on the phone caused by the app and the app actions when the user is using the app, and then with these data the remote server can estimate the power consumption during the logged period for the energy bug diagnosis.

The two trackers on the smartphone are used to collect the runtime information. The utilization tracker is a background service, and the app action tracker is actually the instrumented suspect app done by our Android APK instrumentation tool. When
the suspect app is running, the utilization tracker will log the utilization of system components (e.g. CPU, display, and wireless interface) as well as the information of the system and the app (e.g. system version, app version, and device model); at the meanwhile, the app action tracker will record selected user interactions with the app. Please note that the app action tracker records invoked methods only, and no user inputs or personal confidential data are accessed. Both utilization and action logs will then be sent to the diagnosis agent on the remote server.

After the diagnosis agent receives the logs, it will first estimate the power consumption of the app according to the utilization during the runtime, and at the same time it will infer the action units of the app from the action log. Then the diagnosis agent can estimate the power consumption of each action unit by mapping them to the power consumption estimation results.

The final analysis is done by clustering the estimated power consumptions of action units. The $k$-medoids algorithm [8] is used for the clustering, and two clusters are generated: one is labeled as “normal” and the other is “abnormal.” Because all the data are unlabeled initially and we did not have any presumption against the power consumption of any action unit, we assumed that the larger cluster at the end is “normal,” which implies that most users use a version of the app that does not have energy bugs or abnormal battery drain issues.

Although we implemented our framework prototype on Android-based smartphones, it should still work fine on other platforms. In addition, we did not make any changes to the operating system nor the Android application framework, and developers’ manual effort is basically not required before the diagnosis is done.
2.2 Part I: Online Tracking

2.2.1 Utilization Tracker

The utilization tracker was implemented as a background service on Android, which users can activate manually whenever they have a suspect app running on their smartphone, and the task of this tracker is set to record the utilization of system components on the smartphone for the diagnosis agent at the remote server to estimate the power consumption of the app later.

The smartphone nowadays is usually like a portable computer that is composed of a display, a CPU, memory, storage, network adapter, and so on. In addition, a smartphone has the ability to access the 3G/4G voice and data network, and is usually equipped with other devices, for instance, a camera module and a GPS receiver. Each of the component contributes to the power consumption of a smartphone. In our prototype implementation, only several representative components that usually consume the majority of battery life were chosen to be logged by the tracker: the CPU, the display, and the wireless interface.

The utilization tracker logs the utilization of these components every 500 milliseconds in the following form. For the CPU, it reads the frequency value at time $t_i$, $Freq(t_i)$, and calculates the CPU utilization as follows:

$$U_{cpu}(t_i) = \frac{J_{app}}{J_{sys}} \cdot \frac{\text{CPU time spent on process } pid \text{ during } \Delta(t_i, t_{i+1})}{\text{accumulative total CPU time during } \Delta(t_i, t_{i+1})},$$

where $0 < U_{cpu}(t_i) \leq 1$ and $\Delta(t_i, t_{i+1}) = 500$ milliseconds. In our experiments, we used Google/HTC Nexus One, which is equipped with a single-core CPU; however, this tracker can be applied to a smartphone with a multi-core CPU by reading frequency values of all cores and calculating the utilization for each core. For the
display, it reads two values: (1) the brightness value at time $t_i$, $B(t_i)$, which is an integer value between 0 and 255, and (2) a binary value, $U_{disp}$, which indicates the display utilization (on/off). So the display utilization at time $t_i$ can be presented as:

$$U_{disp}(t_i) = 1, \text{ if the display is on; otherwise 0,} \tag{2.2}$$

For the wireless interface, the tracker simply reads the number of bytes have been sent, $W_{sent}(t)$, and received, $W_{recv}(t)$, so far.

Moreover, the tracker also records the utilization of the suspect app. According to the process identifier (PID) of the app, the tracker can read the information about how much CPU time (jiffies) has spent on the app so far, the number of bytes it has sent or received, and also a binary value, which indicates whether the app is using the display; in other words, the binary value indicates whether the suspect app is running in the foreground. Table 2.1 shows the utilization record generated by the utilization tracker every 500 milliseconds. Besides that, the tracker logs some other information as well, such as the device model, the system version, and the package information of running apps.

### 2.2.2 Action Tracker

The focus of our diagnosis framework is to find out where and how the battery drain issue of an app occurred. From the aspect of the program execution, the energy bug could be one of the two types: one is that it happens all the time when the program runs to that code region, which means it is a deterministic bug; and the other is it happens casually, and usually not everyone running that program would encounter that problem. No matter which type the energy bug belongs to, the trigger for the bug is the same: *there must be something that has been done by the user* before
Table 2.1: The utilization record generated by the utilization tracker every 500 milliseconds.

<table>
<thead>
<tr>
<th>$T_{up}$</th>
<th>$U_{cpu}$</th>
<th>$B_{disp}$</th>
<th>$U_{disp}$</th>
<th>$J_{sys}$</th>
<th>$J_{pid}$</th>
<th>$W_{(sent,pid)}$</th>
<th>$W_{(recv,pid)}$</th>
<th>$D_{pid}$</th>
</tr>
</thead>
</table>

$T_{up}$: system uptime (milliseconds)  
$U_{cpu}$: CPU utilization  
$B_{disp}$: display brightness  
$U_{disp}$: display utilization  
$J_{sys}$: accumulative total CPU time (jiffies)  
$J_{pid}$: CPU time spent on process $pid$ (jiffies)  
$W_{(sent,pid)}$: accumulative number of bytes sent through wireless by process $pid$  
$W_{(recv,pid)}$: accumulative number of bytes received through wireless by process $pid$  

the bug being manifested. Therefore, in order to locate the root cause correctly, it is important to know what did happen. Unfortunately, from the bug reports we saw on popular Android forums or various bug report sites of Android apps, we found out that it is not rare that users could not figure out what they did to cause the problem, and as a result, the bug reports from users usually only describe symptoms and may not be really helpful for developers to fix it, not to mention how to verify the existence of the issue. Figure 2.2, for example, shows a report of battery drain issue [9] on the K9Mail website, which barely provides any useful information to the developers. K9Mail [10] is a popular open-source Email client on the Android platform, and also we used it as the experimental target of our diagnosis framework.

*K9 Mail is a great program. I changed nothing on my phone and Monday morning it started doing this. It runs my cpu constantly, and sends and receives data constantly. This results in drastically increased battery usage, as well as an unknown amount of data usage.*

Figure 2.2: A bug report of battery drain issue on K9Mail, which mainly describes the symptoms with little helpful information for the developers.
To get to know the insight into app actions that triggered the issue, we designed the action tracker to record what users did when they were using the suspect app. We developed an automatic Android APK instrumentation tool, which performs the bytecode-level instrumentation to the suspect app. Instead of instrumenting all the methods in the package, we only targeted those related to user-app interaction events, for instance, screen touch, button click, menu item click, and so forth. The action tracker logs will be used later for the action unit inference and the power consumption mapping.

**Android APK Instrumentation**

![Android APK building process and the instrumentation.](image)

*Figure 2.3:* Android APK building process and the instrumentation.
Most Android apps are written in Java; however, Android does not run the native Java program, and it needs a conversion and packages the converted program into an APK (.apk) file, which is a container that contains all the necessary pieces, in order to run it on an Android device.

Figure 2.3 shows the Android APK building process and the instrumentation. To build an APK file, all the Java source files (.java) will firstly be compiled into the bytecode format files (.class) by the Java compiler. The Android dex tool will then convert the Java bytecode files including third-party libraries into Dalvik bytecode files (.dex). At the end, the apkbuilder tool packages all the Dalvik bytecode files together with all other resources (e.g. images) into an APK file.

Our action tracker implementation targets at instrumenting the APK file, but as you might have learned from the process mentioned above, an APK file is actually a container, and the real program are those Dalvik bytecode files inside the package. Therefore, our instrument process has three steps: (1) unpack the APK file and disassemble the Dalvik bytecode files into “assembly-like” format [11]; (2) perform instrumentation; and (3) compile the instrumented files back to Dalvik bytecode files and then package them back to a new APK file. We used an open-source tool: smali/baksmali [12] to assemble and disassemble the Dalvik bytecode file, and we developed the instrumentation tool based on DroidBox API Monitor [13], which is an open-source Android APK dynamic analysis tool that was designed to monitor Android API calls in an Android app.

Methodology

What we really care is users’ actions; therefore, it is not necessary for us to monitor all the Android API calls or all the method calls. Table 2.2 lists the two main
Table 2.2: Android APIs monitored by the action tracker.

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

categories of Android APIs we monitored. The first category is Android Activity life cycle related methods [14], which indicate the app running status, such as running in the foreground or being killed by the system; the second is user interface (UI) related input events [15], which indicate the interaction between users and UI components (basically everything shown on the screen that the user can interact with). The methods in these two categories usually require developers to override them (life cycle related methods) or to implement them (UI input events) to do appropriate work.

In addition to these methods, the action tracker, if necessary, can be configured to monitor third-party open-source API or other Android APIs that require method
override or implementation. At present, monitoring methods in these two categories is sufficient for the prototype of our diagnosis framework. We instrumented the entry and the exit of these methods, so that we also can know how long the app stayed in each method, and the information will be used by the diagnosis agent later for power consumption mapping. Listing 2.1 shows a log example of K9Mail from the action tracker, where the user composed a new Email when being in the inbox and sent it out.

**Listing 2.1:** Log example of K9Mail from the action tracker. The user composed a new Email when being in the inbox and sent it out.

```
1 1583252 + Lcom/fsck/k9/activity/MessageList; onResume
2 1583253 + Lcom/fsck/k9/K9Activity; onResume
3 1583257 - Lcom/fsck/k9/K9Activity; onResume
4 1583592 - Lcom/fsck/k9/activity/MessageList; onResume
5 1587225 + Lcom/fsck/k9/activity/MessageList; onOptionsItemSelected Compose
6 1587285 - Lcom/fsck/k9/activity/MessageList; onOptionsItemSelected
7 1587286 + Lcom/fsck/k9/activity/MessageList; onSaveInstanceState
8 1587288 - Lcom/fsck/k9/activity/MessageList; onSaveInstanceState
9 1587288 + Lcom/fsck/k9/activity/MessageList; onPause
10 1587289 - Lcom/fsck/k9/activity/MessageList; onPause
11 1587304 + Lcom/fsck/k9/activity/MessageCompose; onCreate
12 1587305 + Lcom/fsck/k9/K9Activity; onCreate
13 1587309 - Lcom/fsck/k9/K9Activity; onCreate
14 1587383 - Lcom/fsck/k9/activity/MessageCompose; onSaveInstanceState
15 1587384 + Lcom/fsck/k9/activity/MessageCompose; onSaveInstanceState
16 1587386 - Lcom/fsck/k9/K9Activity; onPause
17 1587387 - Lcom/fsck/k9/activity/MessageCompose; onPause
18 1608089 + Lcom/fsck/k9/activity/MessageCompose; onFocusChange android.widget.
   EditText@463ce150
19 1608091 - Lcom/fsck/k9/activity/MessageCompose; onFocusChange
20 1611091 + Lcom/fsck/k9/activity/MessageCompose; onFocusChange android.widget.
   EditText@463ce150
21 1611094 - Lcom/fsck/k9/activity/MessageCompose; onFocusChange
22 1619894 + Lcom/fsck/k9/activity/MessageCompose; oncreateOptionsMenu com.android.
   internal.view.menu.MenuBuilder@463c5cd0
23 1619928 - Lcom/fsck/k9/activity/MessageCompose; oncreateOptionsMenu
24 1620833 + Lcom/fsck/k9/activity/MessageCompose; onOptionsItemSelected Send
25 1621016 - Lcom/fsck/k9/activity/MessageCompose; onOptionsItemSelected
26 1621050 + Lcom/fsck/k9/activity/MessageCompose; onPause
27 1621050 - Lcom/fsck/k9/activity/MessageCompose; onPause
28 1621072 + Lcom/fsck/k9/activity/MessageList; onResume
29 1621073 + Lcom/fsck/k9/K9Activity; onResume
30 1621075 - Lcom/fsck/k9/K9Activity; onResume
```
2.3 Part II: Offline Diagnosis

2.3.1 Per-app Power Consumption Estimation

There have been several research work focusing on the power modeling of the smartphone [16–22], and no matter what kind of approach (e.g. component-based or event-based) was used, the linear model was usually adopted to build the model. In our framework, the diagnosis agent uses the component-based linear model in the power estimation module to estimate the power consumption of the suspect app. This module could be built with the utilization tracker on the phone to provide the real-time estimate, but we choose to implement it at the remote server to reduce the overhead on the smartphone. One of the major challenges of the approach is how to achieve the reasonable estimate accuracy with a fairly lightweight online tracker, and from Figure 3.4, our experimental results show that our approach can reach the accuracy of over 85% to 90% in most cases on the Google/HTC Nexus One smartphone.

In the following, we will first describe the way we built the power model for the smartphone, and then how we used it to estimate the power consumption of the suspect app.

Linear Power Model

A smartphone nowadays is much like a combination of a computer and a phone, and therefore, it contains much more hardware components than traditional phones. Some components usually exist in a computer, such as the CPU, the memory, the
storage, and the Wi-Fi interface; some are inherited from traditional phones, for example, the cellular module, the Bluetooth interface, and the camera; and some are unique to smartphones, such as the touch-enabled display and the GPS interface.

In our power model, we chose the CPU, the display, and the wireless interface to be the representative components, and the model of the energy consumption $E$ over a time period $\Delta t$ is shown below:

$$E = \alpha_1(P_{cpu}) + \alpha_2(P_{disp}) + \alpha_3(P_{wifi}) + K_{wifi} + K, \quad (2.3)$$

where $\alpha_{1,2,3}$ are power coefficients, $U_{\{cpu,disp,wifi\}}$ indicates the utilization of each component over the time period $\Delta t$, and $K_{wifi}$ and $K$ are the base energy consumption of the Wi-Fi interface and the phone over the time period $\Delta t$ respectively. We separated the Wi-Fi interface base power consumption from the phone base power consumption in order to have a more accurate estimate. The Wi-Fi interface can be turned off individually, and thus it can affect the value of $K$ if we combine them together. Here $E$ represents the measured whole phone energy consumption when we are building the model for the smartphone, and represents the estimated energy consumption results when we are using the pre-built model with the component utilization of the suspect app.

**Component-based Training**

To consider the effects caused by different component states, we made a training program for each representative component and trained each component by using the approach described below.

1. For each training session, we trained only one component at a time.
2. The training program applies various workload to the component, and the utilization of all other components was kept at a constantly low level.

To train the CPU, the training program set the highest CPU frequency to a specific value that the phone CPU supports and run a large matrix multiplication to guarantee the utilization of the CPU can reach 100% during the training session. We had 7 different levels of frequency values from the highest to the lowest working frequency for the Google/HTC Nexus One. During the training session, we kept the display brightness at the medium and the Wi-Fi interface was kept on but did not have any transmission on going.

For the display, the training program set the brightness to a specific value and kept the CPU and the Wi-Fi interface idle during the training session. The display brightness in Android is defined by an integer value between 0 and 255, and we had 6 different levels set in our training program.

From the results in [16], we learned that the energy consumption cause by the Wi-Fi interface is related to the amount of data transmitted and the transmission rate. The former has a linear relationship with the total energy consumed, and the latter affects the power drawn value, which implies the state of the Wi-Fi interface at that time: the high power state or the low power state. In addition, we also learned that there is no significant difference between the power consumption of the download activity and that of the upload activity. Therefore, the training programs only set the download activities for our modeling, and we used 2 download activities to put the Wi-Fi interface into different power states.
Regression-based Model Building

After we collect all the training data and the measured energy consumption during the training session, we used the linear regression analysis to compute the power coefficients $\alpha_{\{1,2,3\}}$ and the base energy consumption values $K_{\text{wifi}}$ and $K$. The data collected can be presented as follows:

$$
\begin{pmatrix}
E_{1,1} \\
E_{1,2} \\
\vdots \\
E_{1,n} \\
E_{2,1} \\
\vdots \\
E_{k,n}
\end{pmatrix}
= \alpha_1
\begin{pmatrix}
P_{\text{cpu},(1,1)} \\
P_{\text{cpu},(1,2)} \\
\vdots \\
P_{\text{cpu},(1,n)} \\
P_{\text{cpu},(2,1)} \\
\vdots \\
P_{\text{cpu},(k,n)}
\end{pmatrix}
+ \alpha_2
\begin{pmatrix}
P_{\text{disp},(1,1)} \\
P_{\text{disp},(1,2)} \\
\vdots \\
P_{\text{disp},(1,n)} \\
P_{\text{disp},(2,1)} \\
\vdots \\
P_{\text{disp},(k,n)}
\end{pmatrix}
+ \alpha_3
\begin{pmatrix}
P_{\text{wifi},(1,1)} \\
P_{\text{wifi},(1,2)} \\
\vdots \\
P_{\text{wifi},(1,n)} \\
P_{\text{wifi},(2,1)} \\
\vdots \\
P_{\text{wifi},(k,n)}
\end{pmatrix}
+ \begin{pmatrix}
K_{\text{wifi},(1,1)} \\
K_{\text{wifi},(1,2)} \\
\vdots \\
K_{\text{wifi},(1,n)} \\
K_{\text{wifi},(2,1)} \\
\vdots \\
K_{\text{wifi},(k,n)}
\end{pmatrix}
+ \begin{pmatrix}
K_{1,1} \\
K_{1,2} \\
\vdots \\
K_{1,n} \\
K_{2,1} \\
\vdots \\
K_{k,n}
\end{pmatrix}
$$

The model for each component $P_{\{\text{cpu,disp,wifi}\}}$, the Wi-Fi base $K_{\text{wifi}}$, the phone base $K$, and the measured whole phone energy consumptions $E$ for building the power model are shown in Table 2.3. The training session is divided into $k$ 10-second sessions to get the per-session average workload of each component and the base energy consumption, and then the average workload is applied to each tracker record, of which the duration is around 500 milliseconds, that resides in the $k$-th session for the modeling.

Energy Consumption Estimation of the Suspect App

To estimate the energy consumption of the suspect app, we need to collect the component utilization during the runtime of the app, which can be done by our utilization tracker, and with the pre-built power model (Eq. 2.3) we can obtain the estimate. One thing to note is that the models of each component utilization and the base are the same as those used for building the power model in Table 2.3.
Table 2.3: Power model for components, the Wi-Fi base, the phone base and the whole phone energy consumption (for building the power model). The training session is divided into \( k \) 10-second sessions \( \Delta(t_k, t_{k+1}) \) is around 10 seconds) to get the per-session average workload of each component and the base energy consumption, and then the average workload is applied to each tracker record that resides in the \( k \)-th session for the modeling. \( \Delta(t_{(k,i)}, t_{(k,i+1)}) \) is around 500 milliseconds.

<table>
<thead>
<tr>
<th>Component</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>( P_{cpu(k,i)} = C(t_k) \cdot \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( C(t_k) ): the average CPU workload of the ( k )-th 10-second session</td>
</tr>
<tr>
<td></td>
<td>( C(t_k) = (\sum_{1 \leq i \leq n} (F_{cpu}(t_{(k,i)}) \cdot U_{cpu}(t_{(k,i)}))))/\Delta(t_k, t_{k+1}) )</td>
</tr>
<tr>
<td></td>
<td>( F_{cpu}(t_{(k,i)}) ): the CPU frequency during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( U_{cpu}(t_{(k,i)}) \in [0, 1] ): the CPU utilization during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td>Display</td>
<td>( P_{disp(k,i)} = D(t_k) \cdot \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( D(t_k) ): the average display workload of the ( k )-th 10-second session</td>
</tr>
<tr>
<td></td>
<td>( D(t_k) = (\sum_{1 \leq i \leq n} (B(t_{(k,i)}) \cdot U_{disp}(t_{(k,i)}))))/\Delta(t_k, t_{k+1}) )</td>
</tr>
<tr>
<td></td>
<td>( B(t_{(k,i)}) \in [0, 1] ): the display brightness during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( U_{disp}(t_{(k,i)}) ) = 0 ( \lor ) 1 = the display on/off status during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>( P_{wifi(k,i)} = R(t_k) \cdot \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( R(t_k) ): the average data transmission rate of the ( k )-th 10-second session</td>
</tr>
<tr>
<td></td>
<td>( R(t_k) = (\sum_{1 \leq i \leq n} (Data(t_{(k,i)})))/\Delta(t_k, t_{k+1}) )</td>
</tr>
<tr>
<td></td>
<td>( Data(t_{(k,i)}) ) = the amount of data sent/received during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td>Wi-Fi Base</td>
<td>( K_{wifi(k,i)} = W(t_k) \cdot \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( W(t_k) ): the average Wi-Fi base utilization of the ( k )-th 10-second session</td>
</tr>
<tr>
<td></td>
<td>( W(t_k) = (\sum_{1 \leq i \leq n} Base_{wifi}(t_{(k,i)})))/\Delta(t_k, t_{k+1}) )</td>
</tr>
<tr>
<td></td>
<td>( Base_{wifi}(t_{(k,i)}) ) = 0 ( \lor ) 1 = the Wi-Fi on/off status during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td>Phone Base</td>
<td>( K_{(k,i)} = K(t_k) \cdot \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( K(t_k) ): the average phone base utilization of the ( k )-th 10-second session</td>
</tr>
<tr>
<td></td>
<td>( K(t_k) = (\sum_{1 \leq i \leq n} Base(t_{(k,i)})))/\Delta(t_k, t_{k+1}) )</td>
</tr>
<tr>
<td></td>
<td>( Base(t_{(k,i)}) ) = 0 ( \lor ) 1 = the phone on/off during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td>Measured Energy</td>
<td>( E_{(k,i)} = E(t_k) \cdot \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
<tr>
<td></td>
<td>( E(t_k) ): the average phone base utilization of the ( k )-th 10-second session</td>
</tr>
<tr>
<td></td>
<td>( E(t_k) = (\sum_{1 \leq i \leq n} E(t_{(k,i)})))/\Delta(t_k, t_{k+1}) )</td>
</tr>
<tr>
<td></td>
<td>( E(t_{(k,i)}) ) = measured energy consumption during ( \Delta(t_{(k,i)}, t_{(k,i+1)}) )</td>
</tr>
</tbody>
</table>
Table 2.4: Action units inferred from Listing 2.1.

<table>
<thead>
<tr>
<th>ID</th>
<th>Signature</th>
<th>Action Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>e1f6e93cc</td>
<td>+Lcom/fsck/k9/activity/MessageList;--&gt;onResume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Lcom/fsck/k9/K9Activity;--&gt;onResume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/K9Activity;--&gt;onResume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageList;--&gt;onResume</td>
</tr>
<tr>
<td>41</td>
<td>80e7e2bd</td>
<td>+Lcom/fsck/k9/activity/MessageList;--&gt;onOptionsItemSelected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-&gt;Compose</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageList;--&gt;onOptionsItemSelected</td>
</tr>
<tr>
<td>60</td>
<td>4623e3f6</td>
<td>+Lcom/fsck/k9/activity/MessageView;--&gt;onSaveInstanceState</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageView;--&gt;onSaveInstanceState</td>
</tr>
<tr>
<td>30</td>
<td>d22d1f7d</td>
<td>+Lcom/fsck/k9/activity/MessageList;--&gt;onPause</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageList;--&gt;onPause</td>
</tr>
<tr>
<td>42</td>
<td>91d60e76</td>
<td>+Lcom/fsck/k9/activity/MessageCompose;--&gt;onCreate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Lcom/fsck/k9/K9Activity;--&gt;onCreate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/K9Activity;--&gt;onCreate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageCompose;--&gt;onCreate</td>
</tr>
<tr>
<td>43</td>
<td>7a787d19</td>
<td>+Lcom/fsck/k9/activity/MessageCompose;--&gt;onResume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Lcom/fsck/k9/K9Activity;--&gt;onResume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/K9Activity;--&gt;onResume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageCompose;--&gt;onResume</td>
</tr>
<tr>
<td>44</td>
<td>cbaeeebc</td>
<td>+Lcom/fsck/k9/activity/MessageCompose;--&gt;onFocusChange</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-&gt;android.widget.EditText</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageCompose;--&gt;onFocusChange</td>
</tr>
<tr>
<td>40</td>
<td>0e942cf</td>
<td>+Lcom/fsck/k9/activity/MessageList;--&gt;onCreateOptionsMenu</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-&gt;com.android.internal.view.menu.MenuBuilder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageList;--&gt;onCreateOptionsMenu</td>
</tr>
<tr>
<td>46</td>
<td>6e2198a1</td>
<td>+Lcom/fsck/k9/activity/MessageCompose;--&gt;onOptionsItemSelected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-&gt;Send</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageCompose;--&gt;onOptionsItemSelected</td>
</tr>
<tr>
<td>47</td>
<td>1f292834</td>
<td>+Lcom/fsck/k9/activity/MessageCompose;--&gt;onPause</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-Lcom/fsck/k9/activity/MessageCompose;--&gt;onPause</td>
</tr>
</tbody>
</table>
2.3.2 Action Unit Inference

In order to better understanding the appi behavior with the suspect app, we defined action unit as a basic unit that completes some task in the app, for instance, enter the inbox and send a new email.

To infer the action unit, we looked for the pair of the method entrance and exit of each action from the log of the action tracker, and therefore, an action unit could contain only one method call or contain one method call with nested method calls. Table 2.4 shows the action units inferred from Listing 2.1, where we can see the action units involved when the user navigate back to the MessageList (action 26), compose a new Email (action 41 and later) and send it out (action 46 and 47). Therefore, we can now transform a series of app actions into a series of action units.

Power Consumption Mapping

With the inferred series of action units and the energy consumption estimate, the diagnosis agent can calculate the approximate energy consumption of each action unit by summing up the energy consumption during the course of each action unit. Then the diagnosis agent can start the energy bug diagnosis according to the energy consumptions of action units.

2.3.3 Unsupervised Diagnosis

As shown in Figure 2.1, the diagnosis agent has an app database to store app-related diagnosis results, and it can be used to improve the diagnosis. In our current prototype implementation, however, we simply used the unsupervised clustering and labeling technique and assumed that all input data are unlabeled, but we will discuss this potential extension in the later section.
Clustering and Labeling

There have been many clustering algorithms proposed, and the $k$-medoids algorithm is one of the most popular ones, which chooses data points instead of means of data points as centers and is said to be robust to noise and outliers.

Because of our diagnosis purpose, we set the number of clusters $k$ as 2 all the time: one is “normal” and the other is “abnormal.” In addition, we assumed that the majority of users using the suspect app used a version without the abnormal battery drain issue, so the larger cluster in the result will be labeled as “normal,” and the smaller one is labeled as “abnormal.”

The following shows the Partitioning Around Medoids (PAM) algorithm [23], which is an implementation of the $k$-medoids and was adopted in our prototype.

1. Initialize: randomly select $k$ out of the $n$ data points as the medoids.
2. Associate each data point to the closest medoid.
3. For each medoid $m$, for each non-medoid data point $o$,
   
   Swap $m$ and $o$ and compute the total cost of the configuration.
4. Select the configuration with the lowest cost.
5. Repeat steps 2 to 4 until there is no change in the medoid.

The cost in the algorithm can be defined by an arbitrary function that can represent the distance or dissimilarity between two data points.
Chapter 3: Evaluation

3.1 Experimental Setup

In our experiments, we used a Google/HTC Nexus One as our target smartphone, which is equipped with a 1GHz CPU, a 3.7 inch touch-enabled display, and an 802.11 a/b/g interface, and it runs the Android 2.2 (Froyo) system. To measure the phone energy consumption, we used the Monsoon power monitor to achieve fine-grained measurement at 20-millisecond intervals. Most of the analysis implementation were done by Python and R programs.

In the following, we will first introduce our K9Mail test case, and then we will show the accuracy of our power model, and at the end we will present the diagnosis results with the test case.

3.2 Test Case: K9Mail

To verify our proposed framework, we used a bug in K9Mail [9, 24–28] 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. Figure 2.2 also shows one of the bug report that actually did
not provide any useful information to the developers. From this case we can see that it is not so trivial to connect the symptom (e.g. battery drain) to the root cause (e.g. incorrect IMAP settings).

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 increase the number of the “push” folder over the limit. For example, Gmail now has a limit of 15 simultaneous IMAP connections per account [29], and by default one connection is for one folder synchronization activity. Therefore, some users would like to increase the connection numbers to have more folders synchronized at a time without knowing the limit, and the bug will be manifested right after they make the change in the settings of K9Mail.

Once the bug manifests itself, the major symptoms to users are exceptionally high CPU utilization and really slow response from the phone. That is the reason why many users did not know what was going on but only knew that K9Mail consumed lots of CPU resource and drained the battery very fast.

### 3.3 Power Modeling Results

To build the power model for the smartphone, we used the component-based training and the linear regression approach. In this section, we shows the process of the component-based training for the three representative components: the CPU, the display, and the Wi-Fi interface, and then we shows the accuracy of our power model.

Figure 3.1 shows the measured energy consumption and the utilization of each component when training the CPU. The training program set the CPU highest frequency to 7 different levels from the highest to the lowest working frequency, and did
Figure 3.1: Measured energy consumption and component utilization when training CPU.

a large matrix multiplication to make sure the CPU utilization reached 100%. By doing so, the effect to the CPU workload (CPU frequency × CPU utilization) is equal to setting 7 different levels of CPU utilization directly, which is, however, more difficult to control. In addition, the training program maintained the display at the medium brightness, and the Wi-Fi interface was kept on but did not have any transmission. We can see that the energy consumption was basically led by the CPU workload.

Figure 3.2 shows the measured energy consumption and the utilization of each component when training the display. The training program set 6 different levels of
display brightness from the brightest to almost turning the screen off. The CPU was kept with a really low utilization, and the Wi-Fi interface was on but idle. We can see that the energy consumption was essentially led by the display brightness.

Figure 3.3 shows the measured energy consumption and the utilization of each component when training the wireless transmission. The training program tried to trigger the Wi-Fi interface entering different power states by setting different transmission rates, and the display was on but kept at a low brightness value. We can
Figure 3.3: Measured energy consumption and component utilization when training the wireless transmission.

see that although the training program only adjusted the Wi-Fi interface, the CPU was still affected, and the difference in energy consumption was contributed by the changes in both the CPU and the Wi-Fi interface power state.

We used the training data shown above to build the smartphone power model, and we created another 12 test cases, which were involved with various apps, to verify the power model accuracy. Figure 3.4 shows the results over the 12 test cases, which
we compared with the measured energy consumption, and we can see that in most cases our model can achieve more than 85% or even over 90% estimate accuracy.

3.4 Energy Consumption of Action Units

In our experiments, we designed several use scenarios for K9Mail, including account registration, checking new Email, composing new Email and sending it out, browsing Email list and scrolling, modifying settings, and so on. By using the action unit inference, we obtained 59 action units in total, and the average energy consumption per action unit instance is shown in Figure 3.5. The red triangles in the figure
Figure 3.5: Average energy consumption of action units.

indicate those action unit instances happened after an energy bug was manifested, which means those action unit instances were executed with the impact of the energy bug, and we expected that their energy consumption readings were different from those without being affected by an energy bug. We can see that most red triangles have much more average energy consumption than other black points, while those black points usually have lower average energy consumption or have a narrow distribution of the average energy consumption.
3.5 Diagnosis Results

In this section, we show the results of the diagnosis by $k$-medoids clustering. As we described in section 2.3.3, we did not assume any prior knowledge about the input data and the number of clusters was set to 2. We labeled the larger cluster as “normal” because the majority of users should use a version of the suspect app without energy bugs. To evaluate the clustering results, we stored the actual label information of each action unit in advance, and compared them with our cluster labels.

![Figure 3.6: Accuracy of clustering results.](image-url)

Figure 3.6: Accuracy of clustering results.
Figure 3.6 shows the accuracy of the clustering results. We can see that most cases have their accuracy higher than 60%, and about half of them are around or over 80%.

![Chart showing accuracy of clustering results](image)

**Figure 3.7:** Recall of clustering results.

Figure 3.7 shows the recall rates of the clustering results. The recall rate reports the ratio of true positives to all action unit instances that occurred with an energy bug being manifested, and it is an important indicator to show how sensitive our diagnosis approach is. We can see that more than half of the action units in the
figure have at least 60% recall rates, and 9 out of 33 instances reports all the action unit instances with an energy bug manifested correctly.

There are two action units, 3 and 58, having 0 recall rates, and the reason is that those instances affected by an energy bug did not show higher energy consumption from other normal instances as shown in Figure 3.5. This is also the reason for other action units that have lower recall rates.

![Figure 3.8: False alarm rate of clustering results.](image)

Figure 3.8 shows the false alarm rates of the clustering results. It is important for a diagnosis tool to keep its false alarm rate really low because the false alarms might
mislead developers and users and hinder the progress of fixing the issues. From the figure we can see that generally the false alarms rate are pretty low: most of them are under 30%, and 18 over 41 have 0 false alarm rates.

Figure 3.9: Precision of clustering results.

Figure 3.9 shows the precision of the clustering results. The precision shows the ratio of true positives to all the instances reported to have an energy bug, and similar to the false alarms, low precision reports might as well mislead developers and users. From the figure we can see that our diagnosis tool have 18 over 33 cases with 100% precision, but there are two cases have 0 precision, of which the reason, as mentioned
above, is their “normal” and “abnormal” instances could not be differentiated through the samples of energy consumption we collected.
Chapter 4: Extension: Diagnosis with Bayesian Learning

In our current implementation of the diagnosis framework, we only used unsupervised clustering technique to analyze the data and separate them into two clusters, of which the larger one was always labeled as “normal.” Although it seems viable in our experiments, we can still observe several problems in this design.

First, it might not be suitable for all action units to be clustered into two clusters only. For example, those action units with lower recall rates in Figure 3.7 usually have “abnormal” instances with energy consumption too close to the that of “normal” instances, which we called it a grey zone, as shown in Figure 3.5, and it could be better if they were clustered into multiple different clusters. However, without prior knowledge, it is difficult to correctly label multiple clusters with two labels. Therefore, this brings the second problem: blind labeling. The labeling in current design totally relies on the clustering results which are not so reliable in some cases; and thus, it might cause false alarms or false negatives.

To address these problems, we proposed an extension to current design of unsupervised diagnosis as shown in Figure 4.1. At the beginning, all the data are unlabeled, so it works like our current framework. After the clustering and labeling, we report the results to developers or users, and then we wait for their feedback. From their feedback, we gather those confirmed labeled data and create a Bayesian classifier. The
classifier will then output the labeling results with confidence values, and therefore, we will be able to report this information to developers or users as well.

According to the confidence value, users and developers can know how likely an action unit instance suffers from an energy bug compared with others in the suspect app and decide the priority among them. This is helpful for developers especially when there are some grey zone instances, which are expected to have low confidence values, and developers can just skip them temporarily to exam other more confirmed instances instead of spending a lot of time on verifying their bug reports. In addition, the classifier could work better for those with possible multiple clusters because of the prior knowledge of labels.

Figure 4.1: Potential extension of the energy bug diagnosis framework.
Chapter 5: Related Work

5.1 Power Modeling and Estimation for Mobile Devices

Pathak et al. [19] 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. [18] proposed Eprof, a fine-grained energy profiler for smartphones, which based on the finite state machine power model [19] and system call tracing can handle asynchronous energy state and the tail-state energy characteristics of hardware components. Eprof requires modifications to the Android framework.

Yoon et al. [21] 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.

Mittal [17] presented an energy emulation tool that helps developers to estimate the energy use for their apps during development. The tool scales the emulated resources to match those on different mobile devices and provide different operating configurations.

Shye et al. [20] 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 optimizations by reducing the screen brightness over time.

Zhang et al. [22] used the built-in battery voltage sensor and the battery discharge behavior to build a power model, PowerBooster, which requires no external measurement devices. They also developed PowerTutor, an online power estimation tool, by using the model built by PowerBooster.

Balasubramanian et al. [16] 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 Mobile Energy Bug Diagnosis

Kim et al. [30] proposed a power-aware malware-detection framework to monitor and detect previously unknown energy-depletion malwares by analyzing the power signature created from user behavior and power consumption history.

Oliner et al. [31] proposed Carat, a collaborative tool to detect and diagnose code misbehavior that wastes energy on mobile devices. Carat uses a non-invasive app performing coarse-grained measurements on a smartphone and analyzes the correlations between energy consumption and device properties on the server.

Ma et al. [7] proposed a tool called eDoctor that helps general users troubleshoot the abnormal battery drain issue on smartphones. eDoctor captures an app’s time-varying behavior by analyzing its execution phases to identify an abnormal app. In addition, eDoctor gives fix suggestions to users according to the diagnosis results.
Pathak et al. [32] provided a study on wakelock related energy bugs, which prevent a smartphone from going to sleep mode, on Android and proposed a static dataflow analysis approach to detect these bugs. Verkris et al. [33] developed a tool that verifies the absence of wakelock related energy bugs with the inter-procedural dataflow analysis.

Qian et al. [34] proposed ARO, the mobile Application Resource Optimizer, to monitor the cross-layer interaction, such as radio resource channel state, transport layer, application layer, and the user interaction layer, to find out the inefficient of resource usage.

Zhang et al. [35] developed ADEL, the Automatic Detector of Energy Leaks, to detect and isolate energy leaks from unnecessary network communication on mobile apps. ADEL uses taint-tracking technique to trace the direct and indirect use of received data to determine whether it is necessary to the user.
Chapter 6: Conclusion

We have proposed an energy bug diagnosis framework, which analyzes user actions and system component utilization to diagnose the suspect app. We developed an online utilization tracker to collect the system information, and we also developed an automatic Android APK instrumentation tool to instrument the suspect app. An accurate component-based power model has been built to estimate the energy consumption from the utilization log. We further transformed user actions into action units for better understanding app activities. Unsupervised diagnosis was done through clustering and labeling, which shows high precision and low false alarm rates. Our approach does not require system modification nor specific platform supports, and developers manual efforts is barely necessary before the diagnosis results come out. We also proposed an extension to the diagnosis mechanism for better handling multiple clusters and prior knowledge.
Bibliography


[5] “Apple admits there is a problem with iPhone 4S’s battery life - but says it will take weeks to fix.” [Online]. Available: http://www.dailymail.co.uk/sciencetech/article-2056992/iPhone-4S-battery-life-Apple-admits-problem-weeks-fix.html


