DIAGNOSING AND DEBUGGING ABNORMAL BATTERY DRAIN ON SMARTPHONES

BY

XIAO MA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Abstract

The past few years have witnessed an evolutionary change in the smartphone ecosystem. Smartphones have gone from closed platforms containing only pre-installed applications to open platforms hosting a variety of third-party applications. Unfortunately, this change has also led to a rapid increase in *Abnormal Battery Drain (ABD)* problems that can be caused by software defects, misconfiguration, or environmental changes. Such issues can drain a fully-charged battery within a couple of hours, and can potentially affect a significant number of users.

The goal of this thesis is to understand ABD issues, assist smartphone users to diagnose ABD issues and help developers prevent software bugs that may lead to ABD issues. We make three major contributions in different phases of smartphone application development and usage. At the beginning, we study user-reported battery drain issues from major smartphone forums. From this study, we find abnormal battery drain issues dominate user-reported issues, which are presumably more troublesome to users and more difficult for users themselves to diagnose and fix.

The dominance of software energy problems highlights the need for helping app developers to avoid these mistakes. We thus conducted a more thorough analysis on common mistakes programmers make that can introduce software energy problems (e.g., bugs, defects, and inefficient designs). Specifically, we manually examined 117 energy-related software problems in open-source smartphone applications and the Android system. We present common patterns of such mistakes and inefficiencies in the real world, and provide practical implications for developers and researchers. In particular, we discuss the opportunity of using model checking approaches and profiling energy-intensive APIs to detect energy bugs, and present preliminary results.

Motivated by the result, we propose eDoctor, a practical tool that helps regular users troubleshoot abnormal battery drain issues on smartphones. eDoctor leverages the concept of *execution phases* to capture an app's time-varying behavior, which can then be used to identify an abnormal app. Based on the result of a diagnosis, eDoctor suggests the most appropriate repair solution to users. To evaluate eDoctor's effectiveness, we conducted both in-lab experiments and a controlled user study with 31 participants and 17 real-world ABD issues together with 4 injected issues in 19 apps. The experimental results show that eDoctor can successfully diagnose 47 out of the 50 use cases while imposing little power overhead. Although eDoctor is designed to directly help smartphone users, the information collected by eDoctor can also be leveraged by developers to diagnose ABD issues.

To Wanmin, Matt (Sicheng) and Ethan (Sibo)

Acknowledgments

First and foremost, I would like to express my deep gratitude to my advisor, Professor Yuanyuan (YY) Zhou, for initially giving me the opportunity to join the OPERA research group, for pushing me forward while giving me the freedom to go for things that excited me, for all the valuable discussions and countless comments on my drafts, for the willing advice whenever needed, and for the continuing support, belief, and encouragement. I will be forever grateful for having YY as my advisor.

I also consider myself lucky to work with many outstanding collaborators in various research projects, particularly Zhenmin Li, Peng (Ryan) Huang, Xinxin Jin, Soyeon Park, Lin Yan, Ding Yuan and Weiwei Xiong. I wouldn't have been able to finish the research work without their generous assistance in my experiments, discussion on research ideas and encouragement.

I also sincerely thank my other committee members, Professor Chengxiang Zhai, Professor Sam King, and Professor Geoff Voelker, for offering insightful feedback and constructive suggestions on my thesis. My special thanks go to Professor Voelker who has gave me a great deal of help to my paper writing and presentation skills.

The completion of this thesis marks the end of my many years as a student. Among many outstanding teachers I would like to especially thank Guixiang Du and Fenchan Zhang from Wuyi Road Elementary School, Lingxiang Jia from Shanxi Experimental High School and Dr. Fei Wu and Dr. Yueting Zhuang from Zhejiang University, for mentoring me, believing in me, and encouraging me to always aim high.

I am grateful for all the friends with whom I spent my time at UIUC and UCSD, who made the Ph.D. life fun and memorable. In particular, I would like to thank the present and past members of OPERA group, including Weiwei Xiong, Ding Yuan, Yoann Padioleau, Zuoning Yin, Chongfeng Hu, Soyeon Park, Spiros Xanthos, Lin Tan, Shan Lu, Joe Tucek, Weihang Jiang, Qingbo Zhu, Zhenmin Li, Feng Qin, Pin Zhou, Zhifeng Chen, Jiaqi Zhang, Jing Zheng, Peng (Ryan) Huang, Yang (Robert) Liu, Mihn-jong (Michael) Lee, Xinxin Jin, Tianyin Xu, Dongcai Shen, Rishen Chen and Xiaoming Tang. Sincere appreciation is extended to Mary Beth Kelley, Sheila D. Clark, and Virginia McIlwain (UCSD) who helped me in administrative matters.

I would like to express my earnest gratitude to my parents for their support, without which none of my achievements would have been possible. Being the only child of them, and the only child in our extended family who came abroad to study, I feel deeply indebted for their understanding, unconditional support, and countless sacrifices to give me the best possible education. I also thank my parents and parents-in-law for devotedly taking care of my sons while I complete my thesis.

Last, but certainly not least, I dedicate this thesis to my dear wife Wanmin and my beloved sons Matthew (Sicheng) and Ethan (Sibo). They have enlightened my life in so many ways. Words can hardly express how grateful I am for having them in my life.

Table of Contents

List of Tables

List of Figures

1 Introduction

Smartphones have become pervasive. Gartner reports [56] that smartphones accounted for 297 million (19%) of the 1.6 billion mobile phones sold in 2010 worldwide, a 72.1% growth compared to 2009. The momentum continued, as Canalys reported [40] that 487.7 million smartphones were shipped in 2011 — marking the first time that smartphone sales overtook traditional personal computers (including desktops, laptops and tablets).

Configured with more powerful hardware and more complex software, smartphones consume much more energy compared to feature phones (low-end cell phones that provide limited functionality). Unfortunately, due to limited energy density and battery size, the improvement pace of battery technology is much slower compared to Moore's Law in the silicon industry [90]. Thus, improving battery utilization and extending battery life has become one of the foremost challenges in the smartphone industry.

Fruitful work has been done to reduce energy consumption on smartphones and other general mobile devices, such as energy consumption measurement [51, 54, 89, 98], modeling and profiling [61, 84, 98, 104], energy efficient hardware [67, 76], operating systems [50, 53, 57, 75, 93, 100, 101, 103], location services [55, 65, 72, 77], displays [48, 60] and networking [47, 49, 78, 92, 95]. Previous work has achieved notable improvements in smartphone battery life, yet the focus has primarily been on normal usage, i.e., where the energy used is needed for normal operation.

In this thesis, we address an under-explored, yet emerging type of battery problem on smartphones that complements existing work: *Abnormal Battery Drain (ABD).*

1.1 Abnormal Battery Drain Issues

ABD refers to the abnormally fast draining of a smartphone's battery that is not caused by normal resource usage. From a user's point of view, the device previously had reasonable battery life under typical usage, but at some point the battery unexpectedly started to drain faster than usual. As a result, whereas users might comfortably and reliably use their phones for an entire day, with an ABD problem their phones might unexpectedly exhaust their batteries within hours.

ABD has become a real, emerging problem. When we randomly sampled

ID	Category	App/Sys	Root Cause	Resolution
(a)	App Bugs	Facebook	The 1.3.0 release (Aug. 3rd, 2010) of this app contained a bug that kept the phone awake.	Downgrade to the previous version.
(b)	App Bugs	Gallery	The user opened a corrupted picture file in "Gallery", which caused the "mediaserver" process to run into an abnormal state and hog the processor.	Automatically terminate the "mediaserver" the user uses for the "Gallery" app.
(c)	App Config	WeatherBug	A configuration change made "WeatherBug" check locations and update weather information more frequently. Heavier usage of GPS causes the battery to drain quickly.	Roll back the configuration changes.
(d)	App Config	Android Browser	The GPS was continually turned on because the browser was trying to find the location of the user, as requested by "google.com".	Go to "google.com" and disable "Allow use of device location".
(e)	System Bugs	Android System	A bug in the Wi-Fi device driver on Nexus One caused the phone to repeatedly enter its suspend state and immediately wake up, resulting in severe battery drain.	The driver developer has to modify their code to fix the problem.
(f)	System Config	Android System	The user configured the CPU to run at an unnecessarily high frequency.	Roll back the configuration change.
(g)	Environment	Android System	Containing several radiology devices, the office building interfered with cell signals.	Turn on Airplane mode when in the office.

Table 1.1: Representative ABD examples collected from Android forums.

537 real-world cases of user-reported battery-related issues on major Android and iOS forums, we found that more than 90% of them were revealed to be ABD, while only less than 10% were due to normal, heavier usage of resources (for more specifics, refer to Section 2). Further, rather than being isolated cases, many ABD incidents affected a significant number of users. For instance, the "Facebook for Android" application (Table 1.1-a) had a bug that prevented the phone from entering sleep mode, thus draining the battery in as rapidly as 2.5 hours. The estimated number of users for this application was more than *12,000,000* at that time [10], among whom a large portion were likely to have been affected by this "battery bug".

To make it even worse, *ABD issues are difficult for regular smartphone users to diagnose and resolve.* The root causes are mysterious to most users. For example, Figure 1.1 depicts the diagnosis process of the Facebook App bug (Table 1.1-a).

When the users observe rapid battery drain, they first need to use some

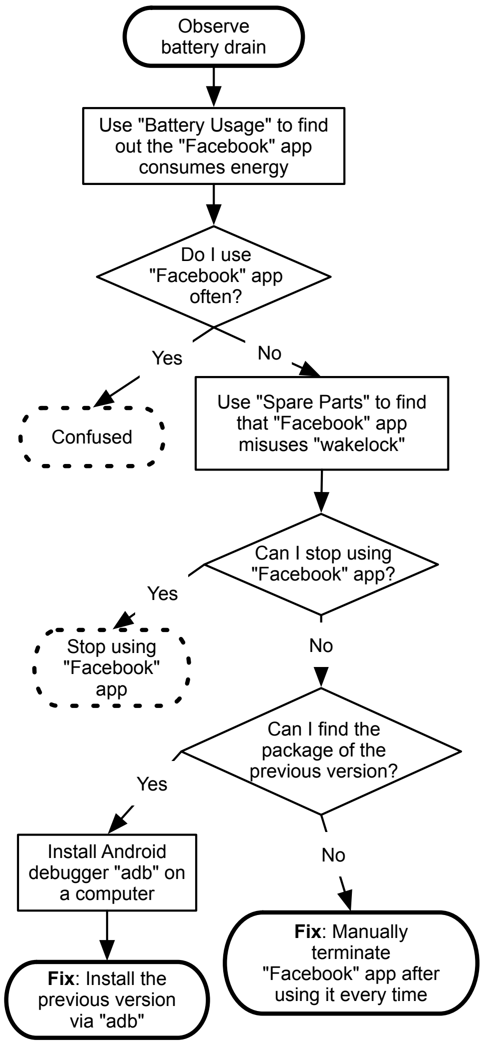

Observe battery drain
Use "Battery Usage" to find out the "Facebook" app consumes energy
Do I use "Facebook" app often?
Yes
No
Confused
Use "Spare Parts" to find that "Facebook" app misuses "wakelock"
Can I stop using "Facebook" app?
Yes
Stop using "Facebook" app
No
Can I find the package of the previous version?
Yes
Install Android debugger "adb" on a computer
No
Fix: Manually terminate "Facebook" app after using it every time
Fix: Install the previous version via "adb"

Figure 1.1: The diagnosis process of the Facebook app bug.

tools to figure out which app causes the drain. “Battery Usage” is an utility that comes with the Android system. It provides high-level information about which app uses more battery. Once the users find the Facebook app is the biggest energy consumer, there could be two scenarios: (1) if the user uses the Facebook app very often, it is normal that the Facebook is the biggest energy consumer; or (2) if the user does not use the Facebook app very often, this information is useful for further diagnosis.

In the next step, users need to understand why the Facebook become a big battery consumer. There are several ways of doing it, but they all require relatively deep knowledge about smartphone systems. For example, users could use a development tool, called “Spare Parts” to further zoom into the battery consuming information. From there, they can find out the reason is that the

Facebook app keeps the phone awake for a long time. In the meantime, the user needs to have the upgrade history in order to decide this symptom only appears with the version 1.3.0. All these information adds up, the user can decide there could probably be a bug in 1.3.0.

As the last step, the user needs to figure out how to fix it. If the user already knows the problem is caused by an upgrade to 1.3.0, she could try to revert it to a previous version, Unfortunately, reverting an app to its previous versions is not supported on Android. However, if the user somehow has access to the old version of the Facebook app, she can use a development tool, called "adb", to install an old version.

Of course the user could also just delete the Facebook app, if she does not need it. However, most users still want to use the app. If they cannot revert it to a non-buggy version, they will have to manually terminate the Facebook app every time they use it, which is very tedious and troublesome for most smartphone users.

To sum up, in order to diagnose and resolve the Facebook app's issue of draining battery, the user will need to use three tools, understand how battery works and how Android manages battery, and have access to an old version of the Facebook app. Apparently this is not feasible to regular smartphone users.

1.2 A Paradigm Shift in Smartphone Industry

The emerging pervasiveness of ABD issues is a collateral consequence of an evolutionary change in the smartphone industry. In the last few years, a new ecosystem has emerged among device manufacturers, system software architects, application developers, and wireless service carriers. This paradigm shift includes three aspects:

(1) The number of third-party smartphone applications (or "apps" for short) has grown tremendously, but their developers can lack sufficient training to be battery-conscious.

A few years ago, smartphones such as BlackBerry ran only applications developed by the smartphone manufacturers themselves, whose developers typically have the appropriate training as well as development and testing infrastructures specifically for mobile devices. In contrast, today's smartphone apps are often developed by third-party or individual developers, and thus the number of apps has grown rapidly: the Android app store has more than 500,000 apps [20] and more than 20 billion downloads [13], and the iOS app store has more than 650,000 apps and more than 30 billion downloads [46]. However, many of these developers tend to focus on features and interfaces, on which app download/purchase decisions are often made. Moreover, most apps are sold for low prices or given away for free, so developers usually cannot afford comprehensive mobile testing utilities and infrastructure. As third-party apps proliferate, battery issues are therefore bound to become more pervasive.

(2) The hardware/software configurations and external environments of smartphones have become diverse.

A modern smartphone involves various hardware manufactures, software vendors, wireless network carriers and a lot of app developers, making it difficult to effectively test battery usage under all circumstances. As a result, many battery-related software bugs escape testing. A real-world example is given in Table 1.1-e, in which an Android upgrade caused serious battery drain on Nexus One phones. Users reported battery drain in as little as four hours. It took the Android team 40 days to find the culprit — a bug in the Wi-Fi device driver. It manifests *only* on certain devices.

(3) Smartphone users become diverse and most users are not technique savvy usres.

More and more consumers adopt smartphones, however, most users are not good at manage such complex devices. Battery issues are especially difficult, as they often require deep knowledge of smartphone systems to diagnse.

(4) In addition to software defects (real-world examples shown as Table 1.1-a,b,d,e), ABD issues can also be caused by configuration changes (e.g., Table 1.1-c, f) or environmental conditions (e.g., Table 1.1-g). In many of such cases, their root causes are not obvious to ordinary users. Therefore it would be beneficial if the smartphone system itself could automatically diagnose ABD issues for users.

Different from laptops, whose software may also have battery-related bugs, smartphones need to be on continuously to receive incoming phone calls or text messages until they are recharged at the end of the day. This characteristic makes the impact of ABD issues much more pronounced for smartphones than for laptops. In addition, users may be more careless when downloading and using smartphone apps because smartphone apps have broader use scenarios but each has limited functionality.

1.3 Contribution

In this thesis, we made three major contributions to solve abnormal battery drain issues on smartphones.

1.3.1 Characteristic Study of Battery Issues on Smartphones

To address this emerging problem, it is important to first understand the characteristics of ABD defects in the real world. Although we have seen anecdotal examples, there are many questions left unanswered. What types of defects are there? How are they distributed in terms of root causes? What are good practices to avoid such defects?

To answer these questions, we first conducted an empirical study of 537 real-world user-reported battery drain issues sampled from five major smartphone forums (Section 2.1). They covered the two most popular mobile platforms, Android and iOS. We developed a taxonomy for these battery drain issues, and identified their distribtuion in the real world. We found that software problems accounted for a significant portion of the battery drain issues (39.2% on Android, 35.1% on iOS) compared to other root causes.

1.3.2 Characteristic Study of Software Bugs That Cause Abnormal Battery Drain

As software problems accounted for a significant portion of the battery drain issues (39.2% on Android, 35.1% on iOS) compared to other root causes, it would thus be beneficial to more thoroughly understand software problems that could cause battery drain. This motivated us to study 117 battery-related software problems in the Android operating system (with about four years of development history) and 29 popular open source Android apps. From them, we characterized common mistakes programmers make that could lead to battery drain.

We found that the examined energy problems can be classified into two simple patterns: resource leaks (40.2%) and resource overuse (59.8%). Resource leaks represent failures in releasing energy-consuming resources (e.g., wakelock, display, GPS, sensor, etc). However, we found that only a relatively small portion of them (11.1%) followed the traditional pattern of missing release calls in certain code paths. Many mistakes were made because of the event-driven programming model on smartphones, which makes automatic bug detection a challenge.

To provide implications for future research in this direction, we further examined what extra information (in addition to the static source code itself) would be required for possible detection. We found that over half (55.3%) of resource leaks need call patterns of event handlers to identify. Furthermore, a large portion (40%) requires call patterns of app-specific event handlers.

Besides resource leaks, 59.8% of the studied cases were mistakes of overusing resources. These mistakes are even more difficult to detect automatically because most of them are specific to certain apps. Resource usage profiling may be of great help in this regard, so we examined what kind of profiling information would be needed for automatic detection and what their distribution was. Based on the results, we provide practical implications for system and app developers.

1.3.3 eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones

We designed and implemented *eDoctor*, a practical tool to help users troubleshoot ABD issues on smartphones. eDoctor runs as a light-weight service on a smartphone to record resource usage and relevant events. It then uses this information to diagnose ABD issues and suggest resolutions. To be practical, eDoctor meets several objectives, including (1) low monitoring overhead (including both performance and battery usage), (2) high diagnosis accuracy and (3) little human involvement.

In order to identify abnormal app behavior, eDoctor borrows a concept, called "phases", from previous work in the architecture community for reducing hardware simulation time [59, 62, 68, 88, 96, 97]. eDoctor uses phases to capture an app's time-varying behavior in terms of resource usage. It then identifies apps that have significant phase behavior changes, and these apps become suspects for the ABD issue being diagnosed.

Unlike prior simulation work that uses the heavy-weight instruction-level methods, e.g., basic block vector (BBV), to classify phases, eDoctor leverages the unique multi-resource (GPS, display, sensors, network, etc.) characteristic of smartphones and proposes two new methods to identify phases, Resource Type Vector (RTV) and Resource Usage Vector (RUV). These methods are better than BBV at capturing phases in terms of resource usage, and they keep eDoctor's power and performance overhead low.

Our experimental results with real smartphone apps and real users show that our captured phases are stable across time and different users. Interestingly, most smartphone apps have only a limited number of common phases (details in Section 4.4.1), further aiding ABD diagnosis.

In addition to using app resource usage to capture phases, eDoctor also records events such as installing new apps, app upgrades, configuration changes, etc. eDoctor uses this information in combination with anomaly detection to pinpoint the culprit app and the causing event, as well as to suggest the best repair solution.

To evaluate eDoctor, we conducted a relatively comprehensive set of experiments including both a controlled user study and in-lab experiments.

User Study

We solicited 31 Android device users with various vendors, hardware and software configurations, and usage patterns, and randomly installed on their own smartphones some popular Android apps with *real-world* ABD issues. These issues were introduced by the original developers, not by us. They cover a wide spectrum of possible ABD issues and various categories of apps, including apps from 12 out of 27 categories on Google Play (previously known as Android Market). Participants ran eDoctor for 7–10 days. From the study, we collected

6,274 hours of resource usage data in total, mixed with normal and abnormal battery usage. eDoctor could successfully diagnose 47 out of 50 cases (94%).

In-lab Experiments

We also measured the overhead of eDoctor in terms of its energy consumption, storage consumption and memory footprint. We used a high-precision power measurement board to measure the overall power consumption of a Nexus One smartphone with and without running eDoctor. The result shows that running eDoctor adds only 1.24 mW of power overhead.

2 Characteristic Study on Smartphone Battery Issues

To gain a better understanding of battery issues that bothered smartphone users, we manually examined user complaints about battery drain and their threaded discussions on major smartphone forums. The reason we chose to study user-reported issues is that these issues are troublesome to users and users cannot resolve them so they had to ask for help on forums.

The goal of this study is to understand what smartphone users complain about battery life. Do users mostly complain about smartphones battery life is shorter compared to feature phones (cell phones that are primarily for making calls and sending SMS messages)? How short battery life could be until it starts bothering users? What are the root causes of battery drain? How do users solve battery drain issues?

2.1 Methodology

2.1.1 Data Sources

According to ComScore's report (August 2011) [8], Android and iOS are the most widely adopted platforms for smartphones, altogether taking about 70% of the market share. Therefore, we chose to study issues from these two platforms. Specifically, we collected issues from three major forums for Android: the Android Central forum [5], the Droid Forum [9], and the Android Forum [6]. For iOS, we studied issues from two major forums: Apple's official support forum [7], and the MacRumor forum [19].

2.1.2 Sampling Method

Each of these forums contained thousands of battery drain issues; manually going through every issue is prohibitively expensive. Thus, we used a sampling methodology as follows for each forum.

2.1.3 Determining Sample Size (K)

In statistics, sample size directly depends on total population. Due to the unstructured textual format in our data sources, the total population of battery drain issues could not be exactly measured. We thus estimated it as follows. We first crawled the subjects and URLs of all threads in the forum (about hundreds of thousands). Then we randomly selected 500 threads to manually digest in order to determine how many of them were actually about battery drain problems. Based on the obtained ratio, we could estimate the total population of battery drain issues in each forum. We then determined the appropriate sampling size to ensure the statistical significance [63] of the overall results for each plaltform.

Selecting Samples

We used Google's site-restricted search to find threads with relevant keywords (e.g., "battery", "drain", etc.) for each forum. We then sifted through the results to find the top K threads (K was the sample size) that were truly about battery drain. More often than not, we found that threads containing the keyword "battery" involved various other matters, e.g., repliers suggesting pulling out battery to resolve certain software misbehavior, or users describing that their battery becoming too warm. Google's search was used here rather than random sampling because it priortized issues/threads that were more relevant, more recent, and likely more searched (pervasive).

For each selected thread, we checked whether it was a resolved case based on the follow-up messages of the victim user. This helped us in conducting root cause analysis (Section 2.2).

Table 2.1 shows the sampling details. The second column shows the number of battery related issues on each forum. This number is estimated by samping. The third column shows the number of issues we randomly sampled from each forum. These issues are battery drain related. The last column shows the number of resolved issues from each forum. The statistic data in the rest of the this chapter are calculated based on resolved issues.

2.1.4 Issue Analysis

We finally identified a total of 537 battery drain issues from the five forums. For each issue, we carefully examined its threaded discussions, corresponding app/system, similar problems reported by other users, and related news reports about the issue if any, all of which together provide us a relatively thorough understanding of its root cause and solution.

We have released the complete dataset at [21]. For each case, it includes the basic information (e.g., post URL, symptoms, successful solutions found, phone model, etc.) as well as our analysis result (e.g., type of solutions and root causes if any).

2.1.5 Threats to Validity and Limitations

Real-world characteristic studies are all subject to a validity problem. Potential threats to the validity of our characteristic study are follows.

Forums	Numb of Battery Issues	Number of Sampled Issues	Number of Resolved Issues
Android			
Android Central	2051	90	71
Android Forum	2856	74	71
Droid Forum	2231	75	72
iOS			
Apple Support	2840	198	101
MacRumor Forum	1442	106	62
Total	11420	543	377

Table 2.1: User-reported battery issues dataset.

Limitations in Data Sources

First, we were not able to investigate battery drain issues on all existing mobile platforms. We chose to focus on two most widely used smartphone platforms that altogether had about 70% market share in 2011. Second, for the two platforms we selected five largest public forums as our data sources. Users may have reported issues elsewhere, but we considered our sources to be generally representative because of their size and popularity. Another limitation of our data sources is that they only contained user-reported cases. Clearly, not all battery drain issues were reported. However, we considered the reported issues to be potentially more severe, bothersome, and challenging for users, which arguably demand more research.

Limitations in Data Samples

We did not exhaustively examine all issues that existed. There were approximately over 11,000 battery drain issues on the five selected forums. Therefore, we used a sampling approach. As shown in Table 2.1, our sample ratios were statistically significant, and our datasets were large enough to be statistically meaningful [64].

Limitations in Manual Classification

Due to the unstructured textual format of battery drain reports, we performed all analysis manually. Almost all manual classification suffers from different degrees of subjective bias. But we made our best efforts to minimize such bias by using cross verification of at least two researchers who had previous experience in studying system defects.

We believe that these limitations do not invalidate our results. At the same time, we urge the reader to focus on overall trends and not on precise numbers.

Root Causes		**Android**	**iOS**
Software Defects	System	22.2%	32.8%
	Apps	17.0%	2.3%
Misconfigurations	System	11.8%	7.5%
	Apps	5.2%	4.0%
Environmental Conditions		6.1%	4.6%
Normal Use		6.6%	4.0%
Hardware Defects		3.3%	5.8%
Other		2.4%	2.9%
Unknown Cause		25.5%	35.1%
Total		100%	100%

Table 2.2: Root causes of user-reported battery drain issues.

We hope that the limitations of our methodology would inspire techniques and processes that can be used to record user reported battery issues more rigorously and with more detailed information.

2.2 What Causes Battery Drain on Smartphones?

Sifting through the 537 sampled issues, we found that 387 of them had an indication that the issue was eventually resolved. For the remaining 150 issues, either the user did not come back to report the end result, or no solution was ever found. We categorized the root causes of the resolved 387 issues into seven categories as shown in Table 2.2.

Finding 1: *Only a small number of user reported issues (4.0% on Android, 6.6% on iOS) are caused by normal yet heavy usage of the phone, e.g., users running energy-consuming apps as intended for a long period of time.*

We found that smartphone users rarely complained about battery drain caused by normal usage. We speculate the reason is that users accept the fact that smartphones offer a rich set of features at the price of shorter battery life compared to feature phones.

Over 90% of battery complaints were revealed to be abnormal battery drain issues that were unexpected, severe (e.g., emptying battery in a couple of hours), and mysterious. They significantly affected user experience. Most such abnormal battery drain cases were related to software problems.

2.2.1 Software Problems

We use "software problems" to refer to bugs, defects, or simply energy-inefficient designs in smartphone apps and systems that cause abnormal battery drain. App problems are attributed to a particular app, while system problems can occur in the OS, frameworks (e.g., the Android framework), and system services (e.g., data synchronization and notification push).

Finding 2: *Software problems are a significant cause for battery drain (39.2% on Android, 35.1% on iOS). In particular, system software problems account for the most significant portion of the complaints (22.2% on Android, 32.8% on iOS).*

One may think that system software are often well developed, tested, and maintained. But unfortunately, they still cause battery issues. Several recent major updates to Android and iOS have severe energy drain: the Gingerbread upgrade [14], the HTC EVO 4G firmware bug [41], and the iOS 5.0 upgrade [15].

The prominence of system energy problems highlights the challenges in avoiding and testing energy problems in software development. Once shipped, system problems could significantly degrade user experience and affect vendor reputation. This is because (1) with a system-wide impact on all involved apps, system problems often cause severe battery drain, and (2) unlike app-specific problems for which users can kill or replace the problematic app with alternatives, system problems are difficult if not impossible to avoid.

Finding 3: *App problems are more prominent on Android (17.0%) than on iOS (2.3%).*

Android and iOS differ in many ways, but one of the most noteworthy difference, energy-wise, is their distinguishing policies on background-running apps. Android offers great flexibility as it allows apps to run in the background indefinitely with the acquisition of "wakelocks", while accessing all resources as foreground apps do [1]. In comparison, iOS is much more restricted, as it only allows certain kinds of apps to run indefinitely in the background such as audio playing and Voice-over-IP [16]. Further, it only allows a background-running app to access arbitrary resources in a given short period of time. Clearly, the flexibility offered by Android comes at a price of increased vulnerability to resource leak and overuse defects.

Overall, the dominance of energy-related software problems underscores the need to investigate their patterns in greater detail in order to provide useful implications for prevention and detection techniques. This motivated us to conduct a more thorough study specifically on software energy problems. We present our results in Chapter 3.

2.2.2 Configuration Changes

Without sufficient knowledge of how a configuration can impact battery life, users may unknowingly make improper changes that lead to high battery drain. In our study, we found that such lack of awareness was high.

Finding 4: *Configuration changes account for many battery drain complaints (17.0% on Android, 11.5% on iOS), including both system-level (11.8% on Android, 7.5% on iOS) and app-level configurations (5.2% on Android, 4.0% on iOS).*

System-wide configuration changes usually have a higher impact than app-specific configuration changes, because they may affect multiple apps at the same time. Common examples include using high speed networks (e.g., 4G network [2]), enabling background data transmission [42], turning on GPS [45] or bluetooth [44], extending screen time-out, increasing LCD brightness, increasing CPU frequency, etc.

App-specific configuration changes involving energy consumption are more diverse. Common types include enabling data sync or background updates (e.g., Facebook [17], Gallery [12]), increasing frequency of periodic updates (e.g., email [11]), adding workload (e.g., more email accounts [3]), demanding better performance (e.g., faster execution or high accuracy of results), etc.

This indicates that developers, especially system developers, should be cautious when providing configuration flexibility that could possibly compromise battery life. When such configuration parameters are being modified, the system should explicitly inform the user about possible consequences.

2.2.3 Other Causes

Hardware defects (including failed batteries, defective sensors, and malfunctioning chargers, etc.) account for less than 6% of battery drain issues on both Android and iOS, indicating that most issues can be resolved or at least alleviated in software. *Environmental conditions:* Besides internal triggers on the phone itself, external environments (e.g., weak radio, Wi-Fi, GPS signal, network type [89], handover oscillation) cause battery drain as well (6.1% on Android, 4.6% on iOS). *Uncommon:* A small number of issues have less common causes (2.4% on Android, 2.9% on iOS). For example, a power user accidentally terminated a critical system process that managed battery charging. *Unkonwn:* A large number of the issues have undetermined causes (25.5% on Android, 35.1% on iOS). Mostly, the battery drain stopped after system reboot or reset, but the exact cause remained unknown. This highlights the unique challenges as well as opportunities for automatic diagnosis tools that can pinpoint the root cause of battery drain and possibly resolve the issue for users.

This characteristic study reveals that software problems are the dominant cause of user complaints about battery life (39.2% on Android, 35.1% on iOS).

It would be beneficial to dig deeper and understand their characteristics. Unfortunately, discussions posted by smartphone users do not contain enough details on the exact bug or inefficiency defect in the code. Thus, we conducted another empirical study to specifically characterize common mistakes programmers make regarding energy use (we refer to them as "energy mistakes").

2.3 Triggering Events

From the study, we find that ABD issues happen after certain events by the user, e.g., installing a buggy app, upgrading an existing app to a buggy version, changing configurations to be more energy-consuming, entering a weak signal area, etc. Figure 2.1 illustrates the time line of an ABD issue taking place.

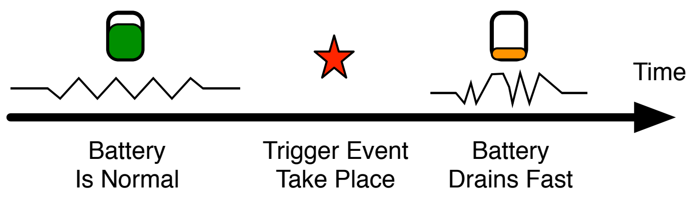


Figure 2.1: Triggering event of ABD issues.

Different triggering events requires different solutions. Simply removing an app is not always ideal. Interestingly, we also find that in more than 60% of the cases, users do not even know the triggering event when the battery drain takes place. In those cases, diagnosing and fixing battery drain becomes even more difficult. Keep in mind that most users who post questions on forums are already tech-savvy users, but they are still puzzled with the root cause of battery drain. It is critical yet challenging for eDoctor to accurately identify the events that cause the battery drain from many other irrelevant events. Only if eDoctor finds the culprit event, it can suggest suitable solutions to users.

Table 2.3 lists the triggering events of the ABD issues in our study. We also calcuate the percentage of each event and the ideal resolution.

Notice that the percentage is calculated based on cases where users know about the events that trigger battery drain, which are only 40% of the total cases we examined. We speculate the remaining 60% of the cases have similar distribution of triggering events.

2.4 Multitasking in Android and iOS

Root causes of battery drain issues have some different characteristics between the two platform we study, Android and iOS. The most significant difference is that iOS has lower ratio of app defects 4.0%, compared to Android where 16.5%

Events	**Percentage**	**Ideal Solution**
App Installation	32.5%	Remove app
App Upgrade	13.3%	Revert to previous version
Configuration change	18.1%	Adjust configuration
Environmental change	14.4%	Adjust configuration
Other	21.7%	Other

Table 2.3: Triggering events and ideal solution of ABD issues.

of the issues are caused by app defects. The reason is that these two platforms have different strategies of running applications in background.

Allowing applications to run in the background provides great flexibility on functionalities, but if many applications run in the background, they may keep the device busy for a long period of time and thus drain battery fast. The situation could become even worse if background applications perform energy intensive tasks, such as querying location information. Android and iOS take different strategies to handle background tasks.

Android

Android gives great flexibility to applications running in the background. They can access all the resources that the foreground application have access to and they can run for long period of time by acquiring wakelocks.

iOS

iOS also allows applications to run in background, but it puts restritions on what they can do. Applications can only do five types of tasks in the background for long period of time:

- Play audio content
- Keep users informed of their location
- Voice over IP - receive phone calls via Internet
- Download newspaper or magazine as a Newsstand app
- Receive periodical updates from external accessories, such as a heartbeat monitor

Apps that implement any these services should explicitly declare the services they support, so the system will prevent apps from being suspended. As we can see, there are many things that background apps cannot do, such as reading data from sensors.

Applications can also execute arbitrary code in background, but they only have a short period of time (e.g., 10 minutes) to finish the work. Developers also need to explicitly implement the code running in background in a special way. The system provides API for developers to query how much time left to execute this code and provides a callback function to be called when the time is close to be used up.

Finding 5: *Allowing apps to run in background brings a trade-off between flexibility and vulnerability of battery drain issues.Android allows background apps to have access to most resources, which provides flexibility on what an app can do; but in the meanwhile, it puts high pressure on battery if many apps run in background and it is more vulnerable to resource leak bugs. On the other hand, iOS has restritions on what an app can do in background. It is less flexible, but it prevents some types of battery related issues.*

3 Characteristic Study on Software Bugs That Cause Battery Issues

To gain a better understanding of battery issues that bothered smartphone users, we manually examined user complaints about battery drain and their threaded discussions on major smartphone forums.

3.1 Methodology

Specifically, we studied the Android Open Source Project (with about four years of development history) and 29 open-source Android apps. Table 3.1 lists our data sources. As shown, we selected apps that covered a variety of categories (e.g., gallery, email client, route tracking), because different types of apps may involve different patterns of energy use. Additionally, we considered only mature apps that had a massive number of users so as to avoid unrepresentative defects in toy programs. For each selected project, we looked into its source code control systems and used keyword search to find commits about energy problems. Since these projects contained thousands of commits, in order to effectively collect energy problems from them, we used a large set of keywords such as "battery", "warm", "energy", "power", "wakelock(s)", "GPS", "sensor(s)", "drain", "accelerometer", "background", "screen", and their variations. We then manually examined the search results, and identified 117 true energy problems to further study.

3.1.1 Limitations

Similar to the previous study, this study has several limitations as well: (1) Data sources. We were unable to cover all open-source apps and systems; however, we believe that our sources are representative because the projects we selected covered most of the categories of Android apps, and we only selected mature apps that had a significant number of users. (2) Data types. The bugs we selected were the ones that had been found and fixed by developers, so we missed issues that were unidentified. We chose to focus on fixed defects for their reliability, because open bugs reported by users may not necessarily be actual energy defects, and may generally be of less priority. Further, we examined 117 issues from 29 apps and the Android operating system, which we believe have covered most representative patterns of common mistakes.

Sources	**Category**	**Description**	**Users**
System			
Android	System	Android system	400M+
Apps			
Android Mail	Communication	Email client	Stock
K9Mail	Communication	Email client	1M+
CSipSimple	Communication	VoIP app	500K+
Linphone	Communication	SoIP phone	100K+
Talking Dialer	Communication	Easy dialer	50K+
Ushahidi	Communication	Information map	5K+
ConnectBot	Communication	SSH client	1M+
Anki-Android	Education	Flash card	100K+
OpenSudoku	Game	Puzzle	1M+
MyTracks	Health	Route tracking	5M+
Gallery	Media	3D gallery	Stock
VLC Remote	Media	Remote control	10K+
MythDroid	Media	MyTV remote	3K+
SongBook	Music	Music management	1K+
Standup Timer	Productivity	Timer	1K+
NanoTweet	Social	Twitter client	10K+
Desk Clock	Tools	Default clock	Stock
Torch	Tools	Torch tool	Stock
SMS Popup	Tools	Messaging	1M+
Marine Compass	Tools	Compass	100K+
Wifi Fixer	Tools	System ultility	100K+
Eyes-Free Shell	Tools	Assistive access	10k+
Mixare	Tools	Argumented reality	10K+
Nice Compass	Tools	Compass	1K+
OpenGPS Tracker	Travel	Route tracking	100K+
BostonBusMap	Travel	Bus tracking	50K+
OpenStreetMap	Travel	Map viewer	5K+
GPSMid	Travel	Vector-based map	150K
funf Open Sensing	Research	Sensor framework	500

Table 3.1: System and apps we collected energy bugs from.

3.2 Overview

The examined energy problems can be classified into two simple patterns: resource leak (40.2%) and resource overuse (59.8%). Resource leak characterizes mistakes that programmers make in releasing a resource (e.g., missing code paths, mis-using resource management primitives). Resource overuse, on the other hand, characterizes mistakes in using a resource.

Type	Percentage
Resource Leak	**40.2%**
Forgetting to release completely or partially	11.1%
Releasing at wrong places	6.0%
Misreleasing in multi-threaded processes	4.3%
Misusing resource management primitives	5.1%
Condition(s) for release unsatisfied	7.7%
Miscellaneous	10.3%
Resource Overuse	**59.8%**
Holding resources longer than needed	29.9%
Using high- instead of low-power abstractions or modes	11.1%
Running jobs more frequently than needed	5.1%
Waking up the phone when not needed	7.7%
Miscellaneous	5.1%
Total	100.00%

Table 3.2: Common energy mistakes programmers make.

3.2.1 Resource Leaks (40.2%)

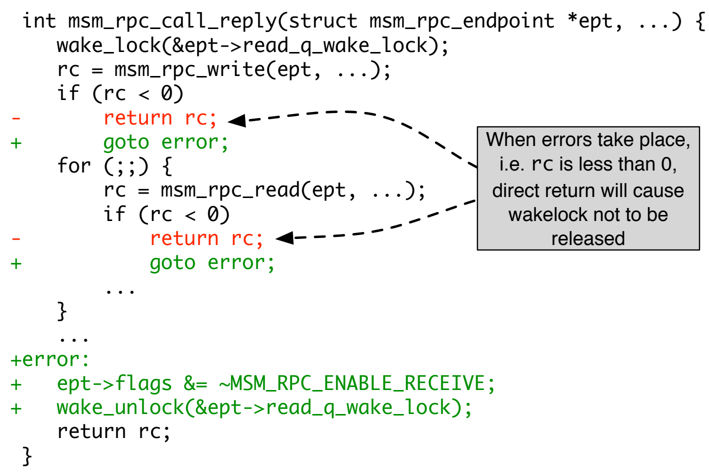
Resource leaks, i.e., failure to release resources (such as wakelock, GPS, and sensors), account for 40.2% of energy defects. When programmers forget to release resources, the device will keep consuming energy without committing meaningful work. The reason why resource leak mistakes are pervasive is two-fold.

First, the current programming model for resource use on smartphones is error-prone. Programmers have to explicitly release resources after use, while acquisition and release could be far away from each other. This is further complicated by the event-driven programming model on smartphones, where resource management is implemented in call-back functions.

Second, resource leak defects do not have immediate symptoms, making them more difficult to test and debug. In order to capture them during development process, developers need to pay significant amount of time to monitor battery usage in various scenarios. Moreover, as we mentioned before, because app purchase is determined by functionality and user interface, developers have little incentive to pay such amount of manual effort to avoid resource leak defects.

We found several common patterns of resource leak mistakes on smartphones.

Forgetting to release (11.1%). Programmers may acquire a resource but completely forget to release it throughout the code. This type of bugs accounts for only 3.4% of all energy mistakes. In other cases, developers remember to release a resource in certain code paths but miss other paths (e.g., error conditions, exceptions). [24] (Figure 3.1) is an example of such type of bugs from the Android system. This type of bugs accounts for 7.7% of all energy mistakes. In the buggy code, there are places where exceptional situation takes place and the system exits the function immediately. However, the correct behavior is to release wakelock first. So the fix is to add error handling code to release wakelock.



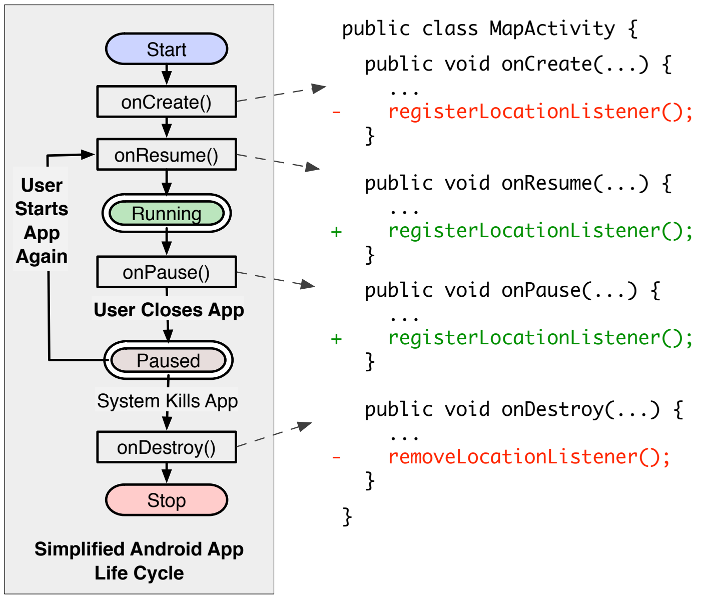
```
 int msm_rpc_call_reply(struct msm_rpc_endpoint *ept, ...) {
    wake_lock(&ept->read_q_wake_lock);
    rc = msm_rpc_write(ept, ...);
    if (rc < 0)
-       return rc;
+       goto error;
    for (;;) {
        rc = msm_rpc_read(ept, ...);
        if (rc < 0)
-           return rc;
+           goto error;
        ...
    }
    ...
+error:
+   ept->flags &= ~MSM_RPC_ENABLE_RECEIVE;
+   wake_unlock(&ept->read_q_wake_lock);
    return rc;
 }
```

When errors take place, i.e. `rc` is less than 0, direct return will cause wakelock not to be released

Figure 3.1: A “Forgetting to release” resource leak bug [24] from Android system

Releasing at wrong places (6.0%). Because of the event-driven programming model, smartphone frameworks let developers implement call-back functions to handle user interaction and system events. It is not uncommon for programmers to place resource management into wrong call-back functions, which may cause battery issues. The bug in Figure 3.2 from the OpenStreetMap app is such an example.

The left-hand-side part of Figure 3.2 shows the life cycles of actitivies in Android. When an activiy starts, three functions are called, `onCreate()`, `onStart()` (omitted for simplicity) and `onResume()`. When the user closes this acitivty, `onPause()` is called. When the user opens this acitivity again, `onResume()` is called to continue this acitivty. So the right places to start and end GPS is `onResume()` and `onPause()` respectively. By doing that, the GPS will not continue working even when the user closes the acitivity. However, the buggy code starts GPS in `onCreate()` and closes GPS in `onDestroy()`, which will not be invoked until the system runs out of memory and the system starts killing background processes. Obviously, this leads to unneccessary GPS usage even if the user closes the app. Since GPS is very energy intensive, leaving it on for a couple hours will drain a battery completely.

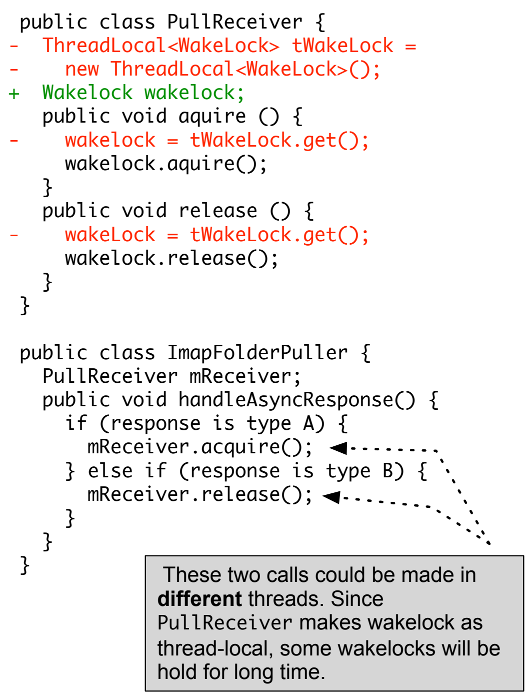


```
public class MapActivity {
  public void onCreate(...) {
    ...
-   registerLocationListener();
  }

  public void onResume(...) {
    ...
+   registerLocationListener();
  }
  public void onPause(...) {
    ...
+   registerLocationListener();
  }

  public void onDestroy(...) {
    ...
-   removeLocationListener();
  }
}
```

Figure 3.2: A resource leak bug in the OpenStreetMap app where the GPS is used in the wrong call-back functions.

Mis-releasing in multi-threaded processes (4.3%). Concurrency programming mistakes also lead to resource leak problems. For example, Figure 3.3 shows a concurrency programming bug [70] in K9Mail (one of the most popular email clients on Android) that causes energy issues. In the buggy version, class `PullReceiver` uses a wakelock to keep the phone awake, but it is declared as a `ThreadLocal` object, which means that a thread will always see the same instance of a wakelock in the same instance of the class `PullReceiver`. When class `ImapFolderPuller` uses receivers, its `handleAsyncResponse()` callback function may acquire or release a wakelock through a `PullReceiver` instance. However, they could occur in different threads, depending on which thread is scheduled to handle asynchronized responses. If the thread that releases the wakelock is different from the one that acquires it, it will not release the correct wakelock, causing energy drain. Concurrency bugs such as data races and deadlocks also account for battery drain problems.

Condition(s) for release unsatisfied (7.7%). In some cases, programmers invoke resource-releasing functions under certain conditions, while such condition may never be satisfied. An example is the bug [94] (Figure 3.4). The process acquires a wakelock when the remaining battery capacity (`raw_soc`) drops below a threshold, and it only releases this wakelock when the capacity grows back to the exact threshold. However, in reality, the capacity may skip the threshold, which will cause the wakelock to not be released. The fix is to change the condition "==" on line 4 to ">=", i.e. to release the wakelock when

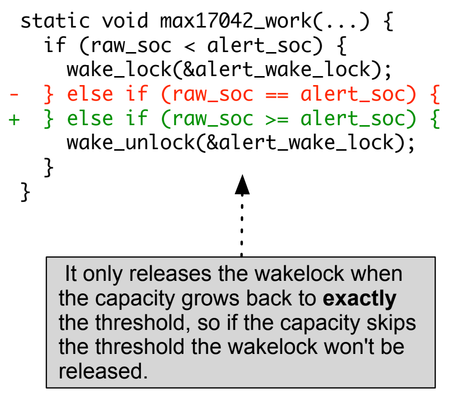


```
 public class PullReceiver {
-  ThreadLocal<WakeLock> tWakeLock =
-    new ThreadLocal<WakeLock>();
+  Wakelock wakelock;
   public void aquire () {
-    wakelock = tWakeLock.get();
     wakelock.aquire();
   }
   public void release () {
-    wakeLock = tWakeLock.get();
     wakelock.release();
   }
 }

 public class ImapFolderPuller {
   PullReceiver mReceiver;
   public void handleAsyncResponse() {
     if (response is type A) {
       mReceiver.acquire();
     } else if (response is type B) {
       mReceiver.release();
     }
   }
 }
```

These two calls could be made in **different** threads. Since `PullReceiver` makes wakelock as thread-local, some wakelocks will be hold for long time.

Figure 3.3: A "Resource Leak" bug caused by ThreadLocal variables.

the capacity is equivalent or greater than the threshold.

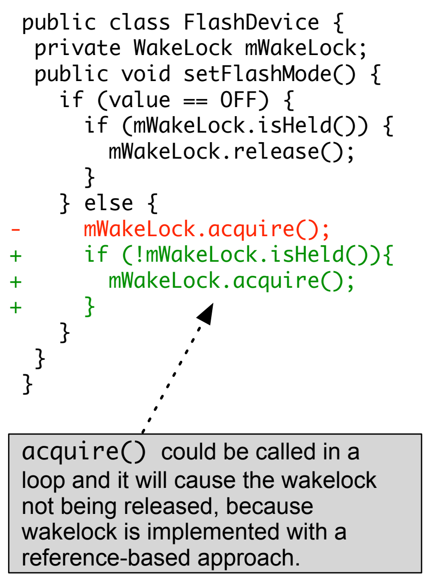


```
 static void max17042_work(...) {
   if (raw_soc < alert_soc) {
     wake_lock(&alert_wake_lock);
-  } else if (raw_soc == alert_soc) {
+  } else if (raw_soc >= alert_soc) {
     wake_unlock(&alert_wake_lock);
   }
 }
```

It only releases the wakelock when the capacity grows back to **exactly** the threshold, so if the capacity skips the threshold the wakelock won't be released.

Figure 3.4: A "Resource Leak" bug caused by conditional errors.

Mis-using resource management primitives (5.1%). Some resource management primitives have deep-level details. If developers are not well aware of them, mistakes are prone to ensue. E.g., wakelock is by default reference-counted in the Android system, and Figure 3.5 shows a bug [58] from the Torch app that acquires a wakelock for multiple times but only releases it once, resulting in a resource leak.



```
 public class FlashDevice {
  private WakeLock mWakeLock;
  public void setFlashMode() {
    if (value == OFF) {
      if (mWakeLock.isHeld()) {
        mWakeLock.release();
      }
    } else {
-     mWakeLock.acquire();
+     if (!mWakeLock.isHeld()){
+       mWakeLock.acquire();
+     }
    }
  }
 }
```

acquire() could be called in a loop and it will cause the wakelock not being released, because wakelock is implemented with a reference-based approach.

Figure 3.5: A “Resource Leak” bug caused by mis-using resource management primitives.

Miscellaneous (10.3%) There are also other miscellaneous mistakes of resource leaks, e.g., the reference to resource objects being overwritten, thus causing a memory leak.

Discussions. The above types of mistakes lead to bugs that are similar to traditional memory leaks. While existing analysis techniques may be leveraged for detection, the event-driven programming model on smartphones makes it a challenge. Under the event-driven model, the execution order of functions depends on external events (e.g. user touch). Pathak et al. [87] studied “no-sleep bugs” and attempted to use data flow analysis for detection. They tried to overcome the challenge of event-driven programming model by manually specifying the calling order of call-back functions in Android framework. However, the coverage was limited because developers often implement their own app-specific call-back functions. Concurrency programming and event-driven nature of smartphone apps introduces challenges to both developers and tool makers. Being familiar with the framework is critical to manage resource in the right call-back functions. Developers should also understand resource management primitives well and use them with caution.

3.2.2 Resource Overuse (59.8%)

In some sense, resource leak represents mistakes in *releasing* a resource. Resource overuse, on the other hand, represents mistakes in *using* a resource. The latter accounts for 59.8% of energy mistakes examined.

Resource Overuse bugs cause devices to keep running for longer than necessary. Similar to resource leak bugs, this type of bugs do not have immediate symptoms, so they often escape from testing. Different from resource leak bugs, this type of bugs have very diverse patterns, which makes it more difficult to detect or debug. Usually fix of these bugs is to release acquired resources more aggressively to preserve battery. We summarized common patterns of this type of bugs.

Holding resources longer than needed (29.9%). This is the most common type of energy mistakes programmers make. There are several cases of such overuse mistakes.

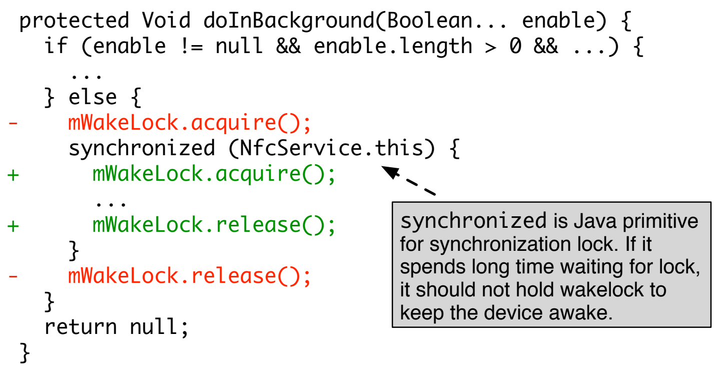
The first case is that resources are held after tasks are finished or interrupted. E.g., the Android system had a bug [35] that keeps sockets alive after communication finishes. Another bug [38] keeps Wi-Fi on when no Wi-Fi connection exists. Normally WiFi will be on for a certain amount of time (e.g., 15 minutes) after the screen is turned off, but if there is no connection, WiFi can be turned immediately to save power. Bug [32] wastes energy by not releasing a wakelock after exceptions occur.

The second case is that a task continues to run after user stops interacting with the phone. E.g., when a user turns off screen, there may be some tasks that are not finished yet. If the result of these tasks will not be delivered to the user, the system or app should preemptively pause or cancel these tasks. For example, a bug [27] in the Android browser keeps rendering web pages even if the browser is switched off. It wastes energy in rendering large web pages or running problematic javascripts.

In the third case, resources are held wastefully during a long wait, such as in synchronization or completion of tasks. Figure 3.6 shows a bug that holds a wakelock while waiting for a synchronization lock which may take a long time. The fixed code grabs NFC Service lock before the wakelock. So even if some process holds the NFC Service lock, at least it does not keep the phone awake while waiting for it. Notice that the snippets are significantly simplified to highlight the important code.

Bug [29] is an example where the Mail app holds a wakelock while waiting for connection to a remove server. It could drain battery if the network connection is slow or the remote mail server has issues.

In the fourth case, resources are held for a long period of time even if the actual resource use is scattered during this period. For example, the bug [28] from MyTrack (a GPS app) holds a wakelock as long as the app is running. This drains battery because the user may record multiple tracks and rest between



```
 protected Void doInBackground(Boolean... enable) {
   if (enable != null && enable.length > 0 && ...) {
     ...
   } else {
-    mWakeLock.acquire();
     synchronized (NfcService.this) {
+      mWakeLock.acquire();
       ...
+      mWakeLock.release();
     }
-    mWakeLock.release();
   }
   return null;
 }
```

synchronized is Java primitive for synchronization lock. If it spends long time waiting for lock, it should not hold wakelock to keep the device awake.

Figure 3.6: A patch that fixes a bug that may hold a wakelock for unnecessarily long.

them. The correct approach is to only hold wakelock when the user is actually tracking his/her route.

Using high- instead of low-power abstractions or modes (11.1%). Due to unfamiliarity about lower-level details, programmers often unnecessarily use high- instead of low-power APIs, protocols, or modes that waste energy. E.g., the email app [26] contained a mistake where high-power replying and forwarding protocols were used where not needed.

For example, the EMail app [26] in the EMail app does not take advantage of the Smart-Reply and Smart-Forward protocols in ActiveSync to save energy. Both of these two protocols downloads and sends less data. With Smart-Reply, only the reply text is sent to the server and the server merges it with original content and compile the final reply text. With Smart-Forward, the user can forward attachments without first downloading to their devices.

Another example is about the usage of the more energy efficient `setInexactRepeating()` function instead of `setRepeating()`. Both functions are used to schedule repeating jobs, but the first one indicates the system can schedule the job at inexact interval time. So the system can group multiple different jobs from different apps and process them all at once. This is especially energy efficient when these jobs need to wake up the device or use network. For example, patch [37] from a Tweeter app on Android adopts `setInexactRepeating()` to save energy.

In system-level code, programmers make the mistake of configuring hardware to run in high- rather than low-power mode when not needed. E.g., [36] sets the brightness of the notification LED light to be overly high, and uses an energy-consuming color. It drains battery when the user does not pay attention to the notification yet leaves it on for a long time. [34] configures the CPU to run at the full speed even when the screen is off. [30] does not enable the lower-power polling feature in the Near Field Communication (NFC) device. [23] configures bluetooth chips to run in the high-power mode while low-power mode is sufficient.

Running jobs more frequently than needed (5.1%). For apps that need to periodically update a status or download information, too frequent updates may quickly drain battery. For example, a defect [31] in Anki-Android updates its widgets at a very high frequency and thus drains battery fast.

Waking up the phone when not needed (7.7%). Sometimes peirodic jobs do not need to run all the time. In particular, for jobs that need to wake up the whole device, it is more appropriate to run repeated instances only when it is necessary. For example, a bug [22] in Android's media library schedules a thread to process time events every 10 ms even if there are no timed events waiting.

Miscellaneous (13.7%). The remaining problems have diverse patterns. E.g., over-optimization on performance causes high energy consumption. Sometimes developers optimize apps to run faster or produce better results, but they are not well aware of the impact on energy consumption. For example, the GPS manager used to need at least 10 calculated positions before disabling GPS, which provides more accurate location information but also consumes more energy [25]. [39] has a data structure intended to optimize performance but makes execution slower on real-world sized datasets and thus drains more energy.

3.3 Implications for Development and Future Research

3.3.1 Automatically Detecting Energy Defects

Effective automatic tools can significantly help developers find defects. We now provide some implications for the future research and development of program analysis tools on smartphone energy problems.

The most promising opportunity is to detect resource leak bugs that have similar patterns as traditional memory leak bugs. Resource leak bugs account for more than a third of all studied energy problems. Although memory leak detection is well studied in the past, the event-driven programming model on smartphone brings unique challenges for resource leak detection. Smartphone apps are primarily implemented as event handler functions, which respond to user interaction and other external events. It is difficult, if not impossible, to construct complete call graphs, which are required by most static analysis techniques.

We will discuss two static analysis approaches that may appliable to detect resource leak bugs.

Dataflow Analysis

Pathak et al. [87] made the first attempt to use data flow analysis to detect resource leak bugs. They tried to overcome the challenges of the event-driven programming model by manually specifying some call patterns of event handlers in the Android framework. However, the coverage was limited because developers often implement their own app-specific event handlers. In that case, it allows developers to provide some hints on how their event handlers will be called. An alternative approach is to collect dynamic execution traces and automatically extract the call patterns of event handlers.

Model Checking

Model checking can *systematically* test and detect resource leak errors in smartphone apps. It enumerates possible states of the app under test by exploring non-deterministic events. Because the explored states are realistic, model checking does not report false alarms. It also have high coverage since it explores as many states as possible. Moreover, it is efficient because it runs automatically without developers to manually drive the checking process. However, there are three major challenges that make model checking smartphone apps difficult.

The first challenge is the cost of abstracting models from apps, which is required by traditional model checking. This process of manual abstraction is not only time consuming, but also error prone. It is very difficult to convince smartphone app developers to do so. To eliminate the cost of model abstraction, a realistic model checking tool should directly apply model checking to the implementation of the app under test. Similar approaches have been proved to be effective in previous work [79] [102] [80]. One could utilize a model checking framework for Java programs, Java Pathfinder [18]. It runs *unmodified* Android apps *in situ* and models system states with actual runtime status of the system. Running Android apps in a model checker is challenging because Android apps highly depend on the Android Framework and native libraries on Android devices. We need to model all the low-level compnent, to give an illution to apps under test as they run on real Android devices. Figure 3.7 depicts the idea of running Android on JPF.

Part of the Android framework is implemented as native libraries (written in C or C++), which cannot be directly ported to run on JPF. We divide native libraries into three categories and port them in different ways.

For native libraries that we need to intercept their execution, for either modeling the system or checking the properties, we write the same implementation in Java so they can run in JPF VM (Figure 3.8 a).

For native libraries that do not affect the system model, we keep using the original implementation. To do that, we need to convert Android style JNI code into JPF style JNI. In JPF, the system under test needs to go to two Java virtual machine to get to native code. So we create Proxies in the host JVM.

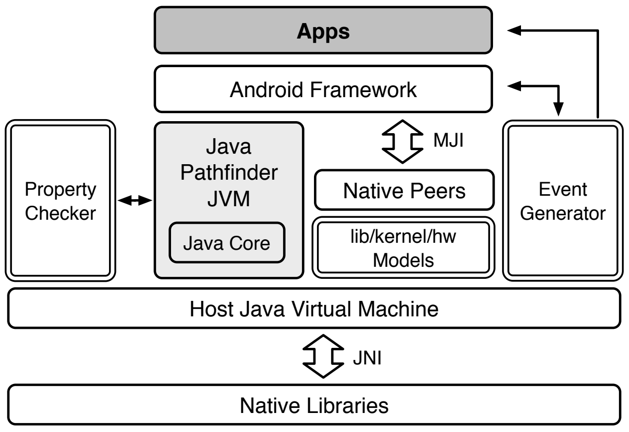


Figure 3.7: High-level design of running Android apps on the Java Pathfinder model checker.

JPF VM first delegates the calls to the proxy and then the host JVM further delegates the class to the native code (Figure 3.8 b).

For native libraries that are mainly for user interaction, we create Java stub to intercept them (Figure 3.8 c). For output to the user, such as rendering the screen or playing sound, we don't need to model., because they are not relevant to the checking. For input from the user, we use the "event generator" to enumerate possible events that drive the execution of the app.

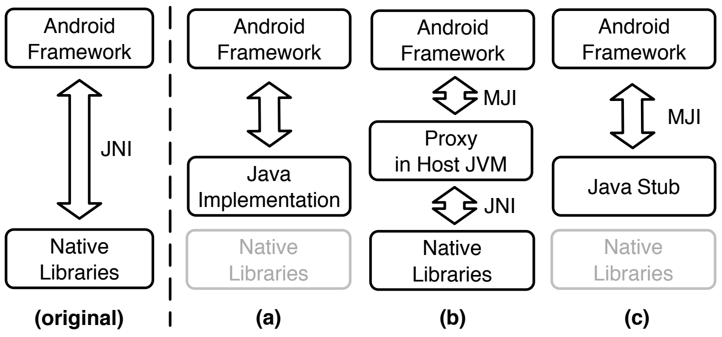


Figure 3.8: Three ways to port native implementation of Android Framework.

The second challenge is how to model user-interactive event-driven systems. The way of modeling a system, i.e., defining states and transitions, has significant impact on the size of the state space. If the states are too fine-grained, it will make the checking unrealistic in terms of both time and memory consumption; on the other hand, if the states are too coarse-grained, the checking may miss mistakes.

The third challenge is the problem of state explosion, i.e., the states of the system becomes too many to be explored completely. This is an inherited issue of model checking. A variaty of optimizations can apply, for example, it should group different components into component groups and make sure interactions between different groups do not affect the properties under check. So it can check groups separately thus eliminate unnecessary states generated by uninteresting interactions.

Static Analysis Feasibility

To facilitate future research and tool development, we analyzed the patterns of resource leak bugs in terms of what extra information is required for possible detection *in addition to* the static source code itself. The result is presented in Table 3.3.

Information Required	**Percentage**
Call patterns of app-specific event handlers	40.4%
Call patterns of system/framework event handlers	14.9%
Thread scheduling	10.4%
Runtime external input	19.2%

Table 3.3: Information needed for resource leak detection.

As one can see in the table, most resource leak bugs require runtime information to be identified, which indicates that static analysis on code paths itself is not enough. Some bugs even require information in multiple categories.

In particular, 55.3% of the resource leak bugs need call patterns of event handlers. Interestingly, we found that most bug-related event handlers are app-specific, instead of general event handlers whose behaviors are pre-defined by the system framework. Manually specifying behaviors of all app-specific event handlers as in [87] may be troublesome for developers.

3.3.2 Identifying Resource Overuse by Profiling

Different from resource leaks, where resources are not released correctly, resource overuse is more difficult to identify by automatic tools. This is because the judgement on whether an app overuses resources highly depends on the semantics of the app, i.e., what it intends to do. In this case, resource profiling information may be greatly helpful for developers to detect resource overuse.

We examined 70 resource overuse defects to understand what kind of profiling information will be useful for automatic detection. The result is presented in Table 3.4.

As one can see in the table, profiling wakelock usage is required by 35.7% of the cases. Wakelock-related issues are often more severe because it keeps the whole phone or most energy-consuming components (e.g. display or Wi-Fi) on.

Required Profiling	**Percentage**
Wakelock	35.7%
CPU	17.1%
Network	14.3%
GPS	10.0%
Sensor	2.9%
Other Hardware Components	20.0%

Table 3.4: Components needed to profile for resource overuse detection.

It is also important to profile usage of other commonly consumed hardware components (such as CPU, network, and GPS), which contribute to 10.0% - 17.1% of the cases.

Besides commonly used hardware components, 20% of the resource overuse cases are related to other devices, including audio device, digital-to-analog converter (DAC), NFC devices, bluetooth, etc. These devices are usually not directly programmed by app developers; instead, they are managed by lower-level system software. So profiling usage of these hardware components may be critical to system software developers in smartphone manufacturers, but not as critical to regular app developers.

There are already profiling tools that can analyze execution traces to optimize energy usage. For example, Application Resource Optimizer (ARO) from AT&T and Instruments from Apple

ARO is an open source tool developed by AT&T. ARO records various usage information on a smartphone and a desktop tool analyzes the log. It is especially good at profiling network usage and reduce energy used for data transmission. It analyzes app behavior against 12 best practices and gives developers suggestions to optimize network usage. For example, it detects duplicated content downloaded through network and suggests better caching mechanism. It identifies scattered data transmission and suggests to transfer data with multiple connections simultaneously. ARO also finds small periodical data bursts that keep the device awake and suggest to group them together or transfer data less frequently. Instruments is a tool from Apple developers can use to analyze battery usage on iOS devices. Once enabled, iOS devices log data about battery usage, including network traffic, processor utilization, GPS usage etc. Developers can import the logs to Instruments and analyze to see if energy is used inefficiently.

3.3.3 Profiling Energy-intensive API Calls on Android

ARO mainly focuses on profiling networking operations to optimize energy usage. Besides networking, other application behaviors may also be energy consuming, and it is important to optimize in other aspects to save energy, for example, GPS and sensors are both energy consuming. Some platforms support keeping the device awake (e.g., wakelocks on Android), which could drain energy fast.

One of approaches is to profiling other intensive battery usage is to track API calls that may use a lot of energy. Directly tracking API usage have several benefits. First, it is hidden from developers, as developers can use those API as usual. Second, it provide a great deal of semantic information about how these APIs are used and why they are used in that way. This information is useful for developers to reason inefficient usage.

In this section, we will discuss an attempt of profiling energy intensive API usage on Android. Similar approaches could be applied to other platforms as well.

There are various ways to profile API usage:

- Asking developers to annotate energy-consuming API calls. The problem of this approach is that it requires a great deal of human effort.
- Instrumenting source code or binary to track API usages.
- Logging usage of APIs in their own implementation. The advantage of of this approach is that it does not require to change either source code and binaries. Even if the developers implement their own primitives to manage these resources, they will call the low-level APIs in the framework. So it gives us the biggest flexibility and compatibility.

In our case, we choose the third approach, thanks to the fact that Android is open source. Android's core application framework is also open sourced, we can directly modify the API implementation, add logging to track their usages, build the modified framework into an image, which can be used by either the emulator or on a real Android device.

For each API, we record the time when it is called, from which process and thread it is called and the hashcode of the object if this API is called by a user-created object (e.g., a method of an `WakeLock` object). Some APIs also have their own specific information to log. Table 3.5 lists all the modified and tracked APIs and additional information we log for each API.

Useful Information Provided by Profiling

To evaluate the prototype of our profiler, we downloaded 26 free apps from Google Play, sampled from 25 ranked apps across 7 categories: Business, Music & Audio, Social, Games, Media & Video, Books & Reference, and Tools. We install the modified framework on an Nexus One device and run each app and enumerate its popular features. Then we analyze and visualize the logged data.

Figure 3.9 shows the CDF of all observed wakelock usage. As we can see, most of the wakelock usage holds wakelock for a short period of time. In 60% of

Modified API	Logged Information
`PowerManager.WakeLock.Constructor()`	Wakelock tag, type
`PowerManager.WakeLock.acquire()`	-
`PowerManager.WakeLock.release()`	-
`ocationManager._requestLocationUpdates()`	-
`LocationManager.removeUpdates()`	-
`SensorManager.registerListener()`	Sensor type, parameters
`SensorManager.unregisterListener()`	Sensor type
`Activity.onCreate()`	Activity ID
`Activity.onStart()`	Activity ID
`Activity.onResume()`	Activity ID
`Activity.onPause()`	Activity ID
`Activity.onStop()`	Activity ID
`Activity.onDestroy()`	Activity ID

Table 3.5: Modified APIs and recorded information.

the logged usage wakelock is held for less than 4 seconds. In 80% of the logged usage wakelock is held for less than 40 seconds. Only in rare cases wakelock is held for more than two minutes. The maximum holding time we observed is less than three minutes. This indicates that very long wakelock holding periods are likely to be a wakelock leak.

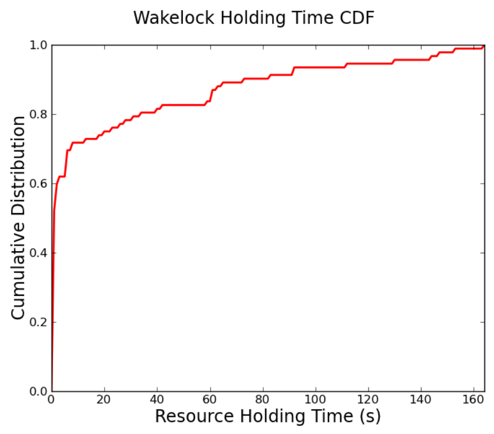


Figure 3.9: Cumulative Distribution Function (CDF) of WakeLock holding times.

Figure 3.10 shows the CDF of number of times an app acquires a wakelock, from all observed wakelock usage. As we can see, most apps do use wakelocks. 40% of the apps only use wakelocks more than 5 times. 20% of the apps only use wakelocks more than 10 times. This indicates that wakelock usage is pervasive although it is prone to energy bugs.

Figure 3.11 shows the CDF of all observed GPS usage. As we can see, most

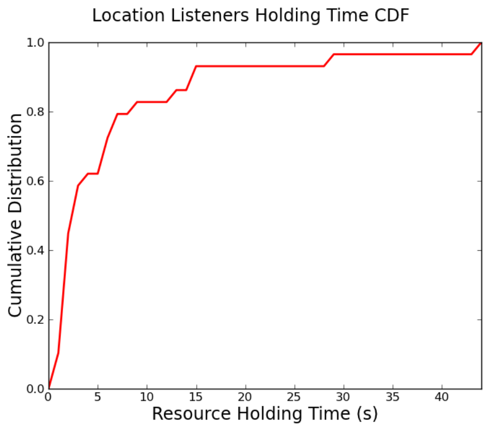


Figure 3.10: Cumulative Distribution Function (CDF) of number of acquisitions of WakeLock.

of the wakelock usage holds wakelock for a short period of time. In 60% of the logged usage wakelock is held for less than 5 seconds. The maximum holding time we observed is less than 50 seconds. This indicates that in most cases, an app uses GPS to get location and shuts it down immediately. The exact usage time period of GPS depends on many other factors in addition to developers' intension, e.g., the strength of GPS signals. There might be cases where GPS is held for a long time, such as GPS tracking apps.

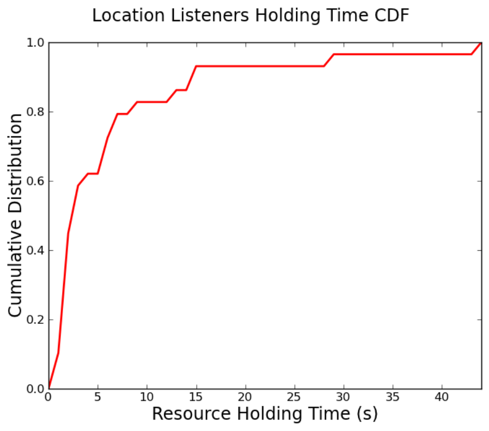


Figure 3.11: Cumulative Distribution Function (CDF) of GPS holding times.

Potential Energy Bugs Detected by Profiling

Besides providing useful debugging information, profiling can also detect potential energy bugs.

Figure 3.12 shows a potential wakelock leak bug in the Google Plus app on Android. As it shows, after we close the app completely, activities are shut down but the wakelock is not released. Also notice in the middle of the testing, there are several periods of time where no activities are active but the wakelock is holding. These time periods may waste wakelock, or they may use wakelock for legitimate reasons. We need further investigation to understand it, but it is out of the scope of this study.

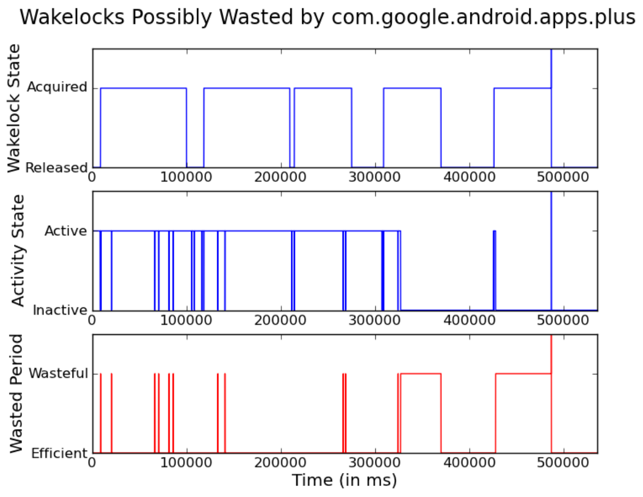


Figure 3.12: A potential wakelock leak bug in Google Plus app on Android.

Figure 3.13 shows a potential wakelock leak bug in the Fruit Ninja app on Android. Similar to the potential bug shown in Figure 3.12, after we close the app completely, activities are shut down but the wakelock is not released. Different from the Google Plus app, the Fruit Ninja app does not have wasted wakelock holding time when it runs.

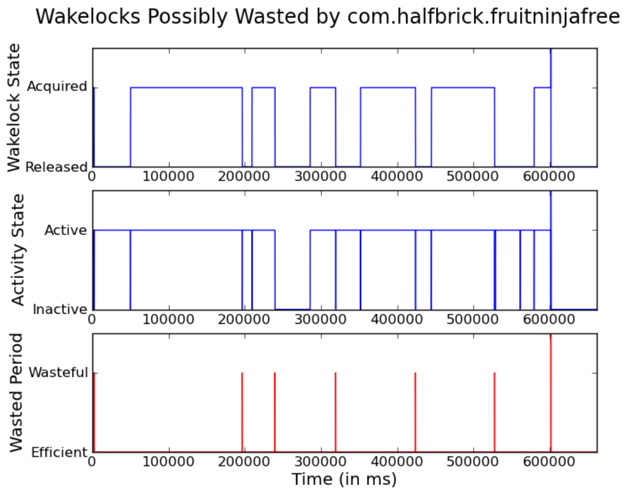


Figure 3.13: A potential wakelock leak bug in Fruit Ninja app on Android.

3.3.4 Energy-cautious Development

Based on the characteristics of energy bugs, we summarize good practices for smartphone app developers as practical guidelines to avoid energy problems.

Rule 1: Make sure to release resources in all possible code paths. Pay special attention to cold code paths, such as error handling code (e.g., Figure 3.1).

Rule 2: Understand application frameworks and APIs. Event-driven frameworks could be complicated with hidden details. For example, the lack of knowledge of the Android's activity life cycle [66] may result in incorrect resource management (e.g., the bug in Figure 3.2); without knowing the reference-counter implementation of wakelock may cause wakelock not being release (e.g., the bug in Figure 3.5).

Rule 3: Be cautious of general programming mistakes when using resources, such as concurrency bugs (e.g., Figure 3.3), missing conditions (e.g., Figure 3.4), object leaks, etc.

Rule 4: Release energy-intensive resources as soon as they are not needed. Sometimes tasks can finish earlier than expected or get interrupted by exceptions. In these cases, resources should be released as early as possible. For example, close network sockets once communication finishes [35], put Wi-Fi component to idle mode if there is no connection [38], and release wakelock once exceptions take place [32].

Rule 5: Manage resource usage in fine granularity if possible. Take advantage of short "no-use" time periods to release resources for preserving energy. For example, do not hold resources while waiting for synchronization locks (e.g., Figure 3.6); do not hold resources while waiting for potentially slow or problematic operations (e.g., [29]); and release resources during breaks in work-flows (e.g., MyTrack [28]).

Rule 6:: Be context-aware and apply different power consumption strategies accordingly. When the device is connected to power supply, an app can take advantage and provide better performance; otherwise, be more conservative (e.g., [33]).

Rule 7:: Execute periodical tasks as infrequently as possible. This is especially important for tasks that needs to wake up certain component of the device or sometimes the whole device, because once a component is turned on, it may take time to go back to sleep even after the task is finished (known as the long tail effect [91]). Reduce frequency, or avoid unnecessary execution if possible.

4 eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones

As the characteristic study on smartphone battery issues shows, ABD issues siginificantly affect user experience. ABD issues can be caused by various reasons, including software defects, configuration changes and environmental conditions. They are usually difficult for smartphone users directly diagnose even if they are tech-savvy users, let along the most smartphone users who do not have the technology background.

So we present *eDoctor*, a practical tool to help users troubleshoot ABD issues on smartphones. eDoctor runs as a light-weight service on a smartphone to record resource usage and relevant events. It then uses this information to diagnose ABD issues and suggest resolutions. To be practical, eDoctor meets several objectives, including (1) low monitoring overhead (including both performance and battery usage), (2) high diagnosis accuracy and (3) little human involvement.

Before introducing eDoctor, we first investigate the state-of-the-art technologies and find they are not sufficient to diagnose ABD issues for users.

4.1 Are Existing Tools Sufficient?

Existing tools (known as energy profilers) such as Android's "Battery Usage" utility, PowerTutor [104] and Eprof [84, 83] can monitor energy consumption of smartphones. While these tools can provide some level of assistance to developers or an elite fraction of tech-savvy users in troubleshooting ABD issues, they are still insufficient for broadly addressing ABD issues due to four main reasons:

(1) The profilers *cannot differentiate normal and abnormal energy consumption.* Even if an application consumes a significant amount of energy, it may not be the root cause of ABD. It may simply be one of the apps most frequently used by the user. To determine whether an app's battery consumption as shown by a profiler is "normal" or "abnormal", a user needs to know the app's resource usage pattern. However, it is unlikely that a typical user would know how much battery an app is supposed to consume, especially since an app's battery usage can fluctuate day-to-day even with normal usage.

(2) The information provided by these tools requires technical background to understand. Some profilers may provide detailed information that helps diagnose abnormal battery drain, but technical terms like “cell standby”, “phone idle”, “wakelock” or “CPU” are confusing for average users to understand. Thus it is still difficult for them to find the root cause.

(3) Even for tech-savvy users, these tools provide only limited clues that are far from sufficient for identifying the *causing event* (e.g., a configuration change, an app upgrade, etc.) that caused an ABD issue. Information about causing events is critical to determine the best possible resolution, such as rolling back to a previous configuration value, uninstalling the offending app, etc. Although uninstalling or stopping a culprit app may fix some ABD problems, doing so is probably not desirable for users. For instance, in the example shown in Table 1.1-a, many users may still want to continue using the Facebook app, so a better alternative is to temporarily revert the Facebook app to a previous version until its developers fix the ABD bug.

(4) As mentioned in Section 1.1, sometimes an ABD issue may be caused by the underlying OS, thereby affecting all apps. In this case, these profiling tools may not be able to shed much light on the root cause, much less be helpful to identify a resolution to an ongoing ABD issue.

There are also tools (e.g., JuiceDefender [73]) that help users make configuration changes to extend battery life, such as offering different energy consumption profiles or location-aware Wi-Fi control. They help preserve energy during normal usage, but they cannot prevent ABD issues or help troubleshoot ABD issues.

Pathak et al. [85, 86] proposed a static analysis tool that helps developers detect non-sleep bugs in source code. Chapter 5 discusses their work in details. Our work is different but complementary, as we focus on helping users directly to diagnose ABD issues caused by various reasons, including software bugs and configuration changes.

From a user’s point of view, a highly desirable solution is to have the smartphone itself troubleshoot ABD issues as automatically as possible, i.e., perform self-diagnosis and suggest solutions with minimum user intervention. Besides helping end users, such systems can also collect helpful clues for developers to easily debug their software and fix ABD-related defects in their code.

4.2 Execution Phases in Smartphone Apps

To identify the problematic app or system for an ABD issue, it is important to differentiate abnormal from normal battery usage. It is natural to immediately focus on the app that is the top battery consumer as reported by an energy profiler. Unfortunately, as shown in Figure 4.3 from a real case, the situation is often not so straightforward because an app’s rank in the battery consumption report can fluctuate over time. We recorded the battery consumption rank of this app reported by the Android "Battery Usage" utility, once every hour. The first 15 hours is the time period when the app does not have the battery bug, whereas the second 15 hours is the period when the bug manifested.

The challenge is that there is no clear difference between normal and abnormal periods. Thus, energy profile and rank are not reliable indicators for troubleshooting ABD issues. Additionally, Figure 4.3 shows that *changes* in battery consumption or rank of an app are also not accurate indicators for abnormal behaviors for similar reasons.

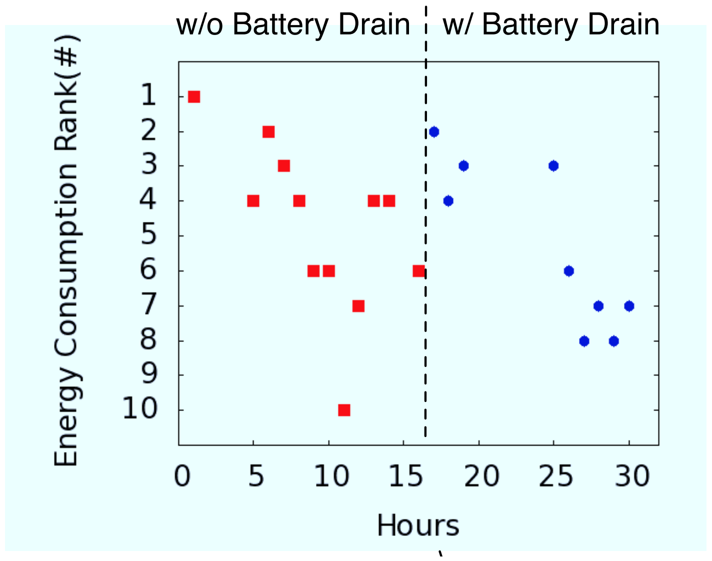


Figure 4.1: Battery consumption rank of the Android Gallery app running on a real user's phone.

In order to identify abnormal app behaviors, eDoctor borrows a concept, called "phases", from previous work for reducing hardware simulation time [59, 62, 68, 74, 88, 96, 97]. This work has shown that many programs execute as a series of phases, where each phase is very different from the others while still having a fairly homogeneous behavior between different execution intervals within the same phase. Hardware researchers simulate those representative phases to evaluate their design instead of the entire execution [97]. Figure 4.2 illustrates the phase behavior of the gzip app. Different color shaded areas present different phases. As it shows, within the same phase, although not consecutive execution, various matrices show similar patterns. In contrast, across different phases, matrices shows different patterns.

Since hardware can observe many execution details efficiently, the above work can use a fine-grained monitoring such as Basic Block Vectors (BBV) to identify phases. For example, BBV is based upon using profiler of a program's code structure (basic blocks) to identify different phases of execution of the program [96].

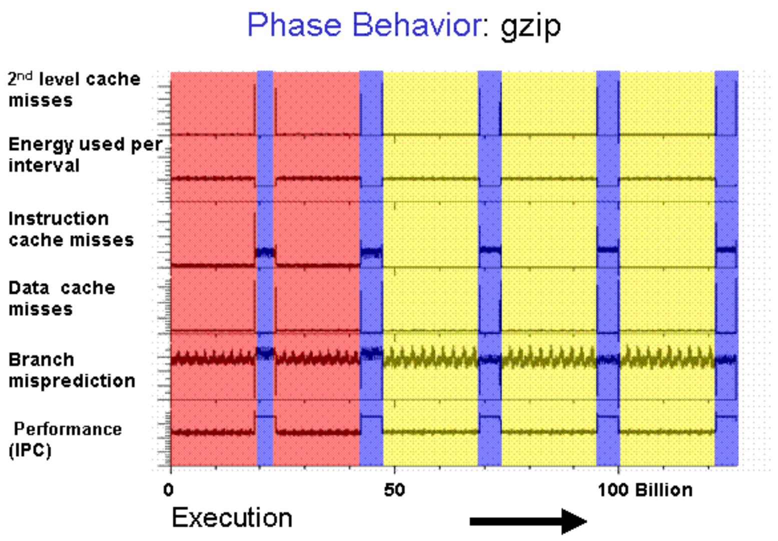


Figure 4.2: Phase behavior of the "gzip" app.

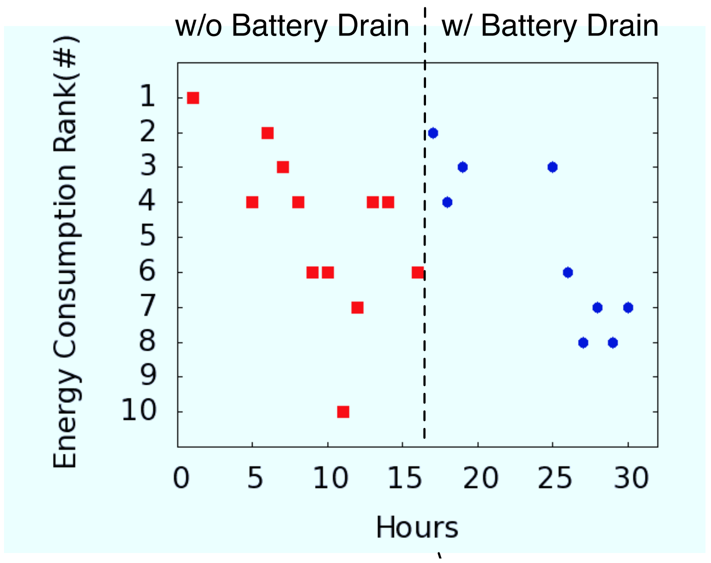


Figure 4.3: Battery consumption rank of the Android Gallery app running on a real user's phone.

4.2.1 Identify Phases in Smartphone Apps

The phase behavior is inspiring for our work because eDoctor can use phases to capture an app's behavior changes in terms of energy usage. When an app starts to consume energy in an abnormal way, its behavior usually manifests as new major phases that do not appear during normal execution. Combining such phase information together with recent events, such as a configuration change, eDoctor can identify both the culprit app and triggering event with relatively high accuracy.

Prior hardware simulation work studied architecture related behavior (e.g., pipeline usage or cache miss ratio). They captured phases based on instruction-level information, such as basic block vector (BBV). However, such fine-grained information is not suitable for identifying phases related to resource usage because instruction-level information does not directly correlate to resource usage. Smartphone apps are different from most desktop/server applications. They are usually relatively simple and not computationally intensive, but rather I/O intensive, interacting with multiple resources (devices) such as the display, GPS, various sensors, Wi-Fi, etc. These resources are energy consuming, so mis-using or over-using these resources leads to battery issues. Therefore, we can capture phases by observing how these resources are used by an app during different execution intervals.

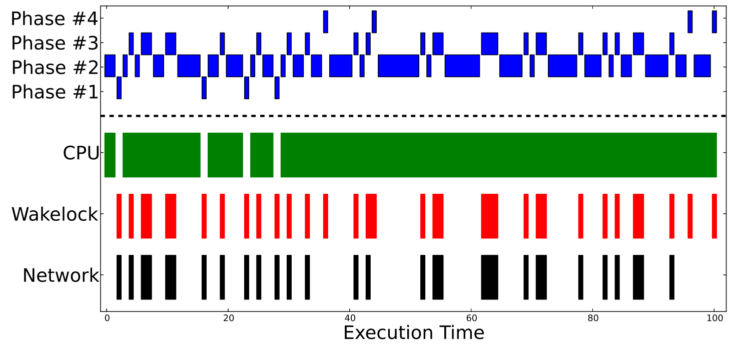
Our first approach starts from a fairly coarse-grain level by monitoring only resource types used during an execution interval, and ignoring the amount of resource usage. We refer to this method as *Resource Type Vector (RTV).* It is based on a simple rationale that different execution phases use different resources. For example, an email client app uses the network when it receives or sends emails. But when the user is composing an email, it uses the processor and display. The RTV scheme uses a bit vector to capture what resources are used in a given interval. Each bit indicates whether a certain resource type is used in this interval. If two intervals have different RTVs, we identify them as two different phases.

As shown in Figure 4.4(a) with data collected from the Facebook app used in a real user's smartphone, RTV clearly shows some patterns and phase behaviors: during different phases, different types of resources are used, and phases appear multiple times during different intervals. As the figure shows, the most common phase is that only the CPU is running. In this phase, most of the time the app is idle. The second most frequent phase has both CPU and network active, which indicates the app transfers and processes data during these phases.

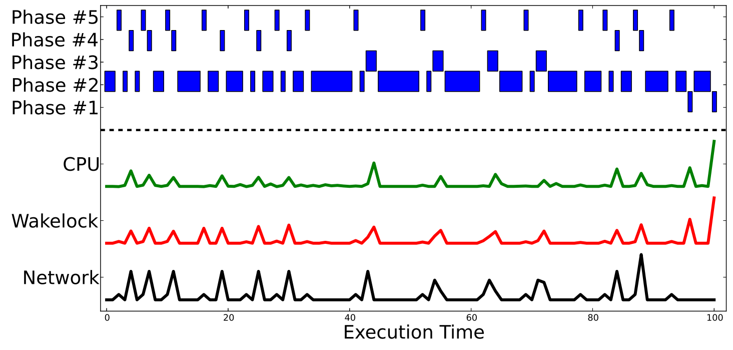
Although the RTV scheme is simple, it turns out to be too coarse-grained since it may merge two distinct phases into one simply because they use the same types of resources. Therefore, we explore a second scheme we call *Resource Usage Vector (RUV).*

Each element in a RUV, represented with a floating point number, is the amount of usage of the corresponding resource. The intuition behind RUV is that an app may use the same types of resources in two different phases, but their resource usage rates differ. For example, for an email app, while both the email updating phase and email reading phase use the display, CPU and network, the resource usage rates are different. The former typically has more network traffic.

We represent the usage of a resource by the usage amount of the resource normalized by the CPU time. The execution interval for measurement cannot



(a) Phase pattern based on RTV. In the top part, the shaded bars indicate which phase the app is in; in the bottom part, shaded bars indicate the resource is in use.



(b) Phase pattern based on RUV. In the top part, the shaded bars indicate which phase the app is in; in the bottom part, the curves indicate the amount of resource usage.

Figure 4.4: The phase behavior of the Facebook App in a real user's smartphone.

be too small to avoid measurement overhead, so an app may run for only a fraction of one interval. In that case, absolute usage numbers cannot precisely represent the usage behavior. CPU time is a good approximation of the amount of time an app actually runs. Normalizing to CPU time allows us to correlate two intervals even if the app runs for different amounts of time in each interval.

If two intervals have similar RUVs, we consider them as one phase. Similar to previous work [97], we use the k-means algorithm to cluster intervals into phases. To find the most suitable k (i.e., the number of clusters to generate), eDoctor tries different k from 1 to 10 at runtime, and for each k, we evaluate the quality of the clusters. We calculate the average inter-cluster distance divided by the average intra-cluster distance as a score; the higher the score is, the better the clusters fit the data. Since the best k is likely to be the largest k it tries, we pick the smallest k whose score is as high as the 90% of the best score.

Figure 4.4(b) shows the RUV phase behavior using the same data. As it

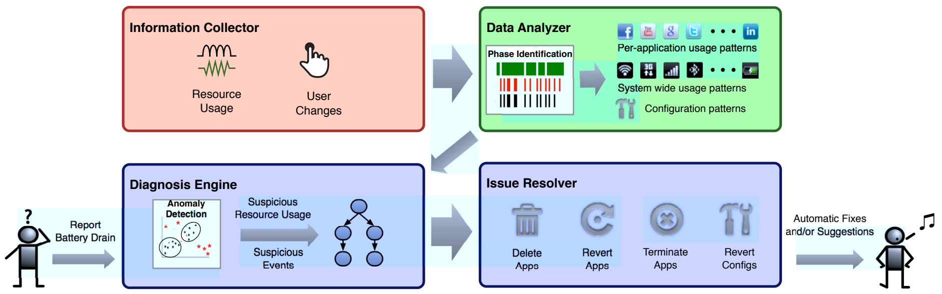


Figure 4.5: Overall architecture of eDoctor.

shows, RUV captures one more phase compared to the phases divided by RTV, enabling RUV to further differentiate between low and high network usage. In other words, it provides more fine-grained information regarding an application's phase behavior.

4.3 eDoctor: Design and Implementation

The objective of eDoctor is to help users diagnose and resolve battery drain issues. Even though the information offered by eDoctor can also be used for app developers, our goal is to help users troubleshoot and/or bypass ABD issues before developers fix their code which as shown may take months. Therefore, instead of locating root causes (e.g., bugs) in source code, eDoctor's diagnosis focuses on identifying (1) which app causes an ABD issue and (2) within the identified app, which event is responsible, e.g., user installed a buggy app, updated to a buggy version, or made an improper configuration chanage. Based on such diagnosis result, eDoctor can then suggest appropriate repair solutions to users, such as uninstalling the buggy app, reverting the buggy app to a previous version, or reverting a configuration change to its previous setting.

There are two major challenges involved in achieving these objectives. First, it is non-trivial to accurately pinpoint which event accounts for the ABD issue. The causing event may not be the most recent one; instead, it can be followed by many other irrelevant events, e.g., the case where the user installed a buggy app and then made multiple configuration settings for this app. Second, eDoctor itself should not incur high battery overhead by collecting too much information too frequently. It needs to balance the overhead and the amount of information needed for accurate diagnosis.

This section presents our design of eDoctor in addressing the above challenges. Figure 4.5 shows the high-level system architecture. First, eDoctor continuously collects system-level and app-level resource usage data, as well as events that may affect battery usage (*"Information Collector"*, Sec. 4.3.1). An event can be of various types, such as configuration change and app maintenance (update, new install). When the phone is idle and connected to a power supply, eDoctor analyzes historical data to learn patterns of resource usage and configuration exhibited when battery consumption is normal (*"Data Analyzer"*, Sec. 4.3.3). When users notice ABD symptoms (i.e., battery draining abnormally fast), they can invoke eDoctor for troubleshooting[1]. eDoctor diagnoses the issues by comparing the recent usage to regularly derived patterns for anomaly detection, and identifying the responsible event (*"Diagnosis Engine"*, Sec. 4.3.3). If applicable, eDoctor provides provides the most relevant repair suggestion to users (*"Repair Advisor"*, Sec. 4.3.4).

4.3.1 Information Collector

Information Collector records three main types of raw data in the background: (1) each app's resource usage, (2) each app's energy consumption, and (3) relevant events such as app installation, configuration, and updates.

The design of information collection faces two major challenges:

- *To minimize the overhead* - Since the information collection runs on smartphones with limited resource, it should minimize the overhead in all aspects, including storage, memory, CPU utilization and most importantly, energy consumption.
- *To collect enough information but avoid over-collecting* – The information collection should be able to collect complete information we need to diagnose battery drains. However, it should also avoid collecting redundant or unnecessary information, which brings overhead.

Our design of information collection is largely guided by real world battery drain issues and the understanding of smartphone systems. It is implemented with minimizing overhead as the first priority.

Resource Usage

eDoctor monitors the following resources for each app: CPU, GPS, sensors (e.g., accelerometer and compass), wakelock (a resource that apps hold to keep the device on), audio, Wi-Fi, and network. To facilitate diagnosis, eDoctor records resource usage in relatively small time periods (called *recording interval*). The default recording interval is five minutes in our implementation.

More specifically, eDoctor periodically collects per-app and system-wide resource usage data at the same time with the same frequency (e.g., 5 minutes), so that they can be correlated in analysis later.

(1) *Per-app resource usage data* collected by eDoctor includes 10 variables: time periods when (the app) holding wakelock (in milliseconds), when running

[1]Currently, eDoctor relies on users to report ABD problems. We leave automatic detection of ABD to future work.

in foreground, when using CPU, when using Wi-Fi, when using GPS, and when using sensors; the amount of sent data (in bytes), of received data; the number of times being launched, and the number of inputs from users.

(2) *System-wide resource usage data* collected by eDoctor includes 13 variables: time periods when the phone is awake (in milliseconds), when Wi- Fi is turned on, when Wi-Fi is in use (actively transmitting data), when CPU is running, when GPS is working, when the screen is on, when Bluetooth is on, when no radio signal is detected, when radio signal is scanned, and when the phone is charging; the amount of sent data (in bytes), of received data (in bytes), and of remaining battery (in percentage).

What resource usage information to store depends on the phase identification method (Section 4.2). RTV uses a bit vector to record whether the resources have been used in the past recording interval (e.g., whether Wi-Fi is on, whether accelerometer is used, whether any data are transmitted through the network). RUV, on the other hand, records the usage amount of each individual resource, e.g., time in microseconds, amount of network data in bytes.

In our implementation, eDoctor takes advantage of the battery usage tracking mechanism in the Android framework. This mechanism keeps a set of data structures in memory to track resource usage of each app. However, the values recorded are accumulated amounts since the last time the phone is unplugged from its charger. At the end of each recording interval, eDoctor reads these values and calculates the resource usage amounts in the past recording interval. Figure 4.6 shows a simplified example of a resource usage table for an app.

Energy Consumption

It is also important to record battery consumption of each app in every recording interval. eDoctor uses this information for two main purposes: (1) to prune apps with small energy footprints which are unlikely a cause for ABD, and (2) to rank suspicious apps according to the energy consumed. As we use the battery consumption information only for such comparative purposes, it is less critical to have high fidelity measurement. Further, simple models provide superior performance benefits that are essential to reduce overhead of eDoctor. Therefore, we employ the efficient profile-based energy model instead of expensive state-based energy models [98, 104]. This energy model has been used in both industry (e.g., Android's Battery Usage utility [4]) and academic research (e.g., ECOSystem [103]).

More specifically, the model is based on resource usage (collected by eDoctor) and average energy consumption profile of hardware components (measured and provided by device manufacturers). Based on the nature of the resource type, resource usage(and the energy consumption profiling) can be expressed in two forms: time-based and quantity-based. The former means how long has a certain type of resource been used, examples include the usage time of CPU, GPS, Wakelock, etc. The latter indicates how much of that type of resource has been used. The usage of Wi-Fi and Mobile data is expressed in this form.

For time-based resource, we take estimation of processor power usage as an example. A processor may work at different frequencies, so energy consumption of processors are profiled on different frequencies. Processor usage data is also collected based on different frequencies. To estimate the power consumption, the model simply multiplies the usage time by the average power consumption of each frequency, and then sum them up:

$$Power_{cpu} = \sum_{f=1}^{n} T_{cpu}^{f} \times AvePower_{cpu}^{f} \tag{4.1}$$

Where n is the number of different frequencies this processor can work on; T_{cpu}^{f} is the amount of time the processor works on the fth frequency; and $AvePower_{cpu}^{f}$ is the profiled average power consumption when the processor works on the fth frequency.

We briefly explain the case of Wi-Fi as an example of how quantity-based resource power consumption is estimated. The energy estimation for Wi-Fi in the profile from the platform is given in unit time. To convert it to quantity-based result, we first extract or estimate the device's Wi-Fi data rate in Bps(Bytes per second). Then the energy estimation is divided by the data rate to get the power consumption per byte. Given the data in bytes received or sent from Wi-Fi, we multiply them as an estimation of the total power consumed by this amount of data transfer.

Every Android-based device has power consumption profile information, which is required to be provided by device manufacturers. This profile provides average power consumption for various hardware components, such as the average energy consumption when Wi-Fi device is turned or when Wi-Fi device is sending/receiving data. These data are measured by high-precision power meters when the phone is made. Each Android system provides a template XML file to store these information (List 4.1 is an example). As seen in the list, the profiling information is relatively complete. For example, it provides the energy consumption of the processor when the processor is running at four different clock speed. The Wi-Fi device energy consumption also has breakdown information for different state of the Wi-Fi device.

Listing 4.1: **An example of a device power profile XML file**.

```
<device name="Android">
  <!-- All values are in mA except as noted -->
  <item name="none">0</item>
  <!-- min brightness -->
  <item name="screen.on">200</item>
  <item name="bluetooth.active">150</item>
  <item name="bluetooth.on">1</item>
  <item name="bluetooth.at">1</item>
  <!-- 360 max on calendar -->
  <item name="screen.full">160</item>
  <item name="wifi.on">1</item>
  <item name="wifi.active">150</item>
  <item name="wifi.scan">200</item>
  <item name="dsp.audio">150</item>
  <item name="dsp.video">200</item>
  <item name="radio.active">150</item>
  <item name="gps.on">55</item>
  <item name="battery.capacity">1750</item>
  <item name="radio.scanning">90</item> <!-- TBD -->
  <!-- Current consumed by the radio at different
       signal strengths, when paging -->
  <!-- 1 entry per signal strength bin, TBD -->
  <array name="radio.on">
      <value>3.0</value>
      <value>3.0</value>
  </array>
  <array name="cpu.speeds">
    <value>350000</value>
    <value>700000</value>
    <value>920000</value>
    <value>1200000</value>
  </array>
  <!-- Power consumption in suspend -->
  <item name="cpu.idle">7</item>
  <!-- Power consumption due to wake lock held -->
  <item name="cpu.awake">20</item>
  <!-- Power consumption at different speeds -->
  <array name="cpu.active">
      <value>120</value>
      <value>228</value>
      <value>299</value>
      <value>397</value>
  </array>
</device>
```

Events

Events are important for both diagnosis and repair advisory. eDoctor records two types of events: (1) configuration changes, and (2) maintenance events (installation, updates). It is worth noting that such events may be initiated not only by the users, but also by the underlying system automatically. App and system configuration entries and their new values are recorded as key-value pairs.

Most apps use Android's facility components (e.g., `PeferenceActivity` and

`SharedPreferences`) to manage configurations. These components provide interfaces such as radio buttons, text input boxes, multi-selection checkboxes, and allow apps to store new configuration values specified by users. They also manage storing configuration values in a centralized location in each app. By placing hooks in these common components, eDoctor track app configurations by modifying these common components.

For system-wide configurations, eDoctor records changes that may affect battery usage, including changing CPU frequency, changing display brightness, changing display timeout, toggling Bluetooth connection, toggling GPS receiver, changing network type (2G/3G/4G), toggling Wi-Fi connection, toggling Airplane mode (which turns off wireless communications), toggling the background data setting, upgrading system, and switching firmware. In the implementation, eDoctor records events by capturing broadcast messages by the Android system. For example, when the Wi-Fi connection status changes, the system sends a broadcast message, `WIFI_STATE_CHANGED_ACTION`.

Location and Environmental Conditions

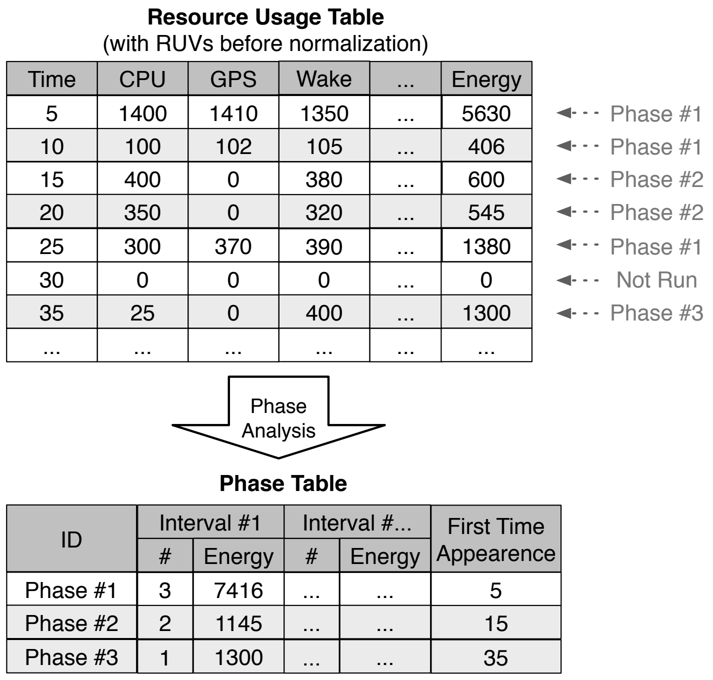
eDoctor records *location change events*, i.e., when the user moves from one geographical location to another. For each location, eDoctor also records the associated *environmental conditions*, including 5 variables: geographical center (in degree), number of appearances, average radio signal strength (in dB), accessible Wi-Fi access point(s), and average Wi-Fi signal strength (in dB). Naturally, such environmental conditions are very similar among close-by geographical positions. To make information collection more efficient, eDoctor merges geographical positions that are close enough as one location.

To protect user privacy, eDoctor stores the above information in its app-specific storage that other apps cannot access. In addition, it does not transfer the information outside of the phone; all analysis is done locally.

4.3.2 Data Analyzer

eDoctor's Data Analyzer is responsible for parsing all resource usage data collected by Information Collector, generating phase information (Section 4.2) for each app, and storing it in a per-app *phase table*. The phase information is useful for speeding up the diagnosis process (Section 4.3.3). Since such phase analysis incurs overhead, it is only performed when the phone is being charged and the user is not interacting with the phone. Figure 4.6 provides a simplified example of how phase analysis is done.

In Figure 4.6, the resource usage table shows seven resource usage records collected by using the RUV method (before normalizing to CPU time). Based on k-means clustering computation (Section 4.2), entries with timestamp 5, 10 and 25 belong to the same phase (Phase #1 in the Phase Table below), because they have similar usage patterns even though the absolute values of their entries differ largely. In addition, the entries at time 15 and 20 belong to the same phase (Phase #2), as the app only uses CPU for data processing (in this simplified example, we assume the values in the other columns on these rows are all zero). The entry at time 30 indicates that the app is not running, so it is not inserted in the Phase Table. The last entry at time 35 is another new phase (Phase #3) where only wakelock is held for a long time but the app does not use much other resources. It is the typical symptom when the app forgets to release wakelock.



Resource Usage Table
(with RUVs before normalization)

Time	CPU	GPS	Wake	...	Energy	
5	1400	1410	1350	...	5630	Phase #1
10	100	102	105	...	406	Phase #1
15	400	0	380	...	600	Phase #2
20	350	0	320	...	545	Phase #2
25	300	370	390	...	1380	Phase #1
30	0	0	0	...	0	Not Run
35	25	0	400	...	1300	Phase #3
...	...	...	...	...	...	

Phase Table

ID	Interval #1		Interval #...		First Time Appearence
	#	Energy	#	Energy	
Phase #1	3	7416	...	...	5
Phase #2	2	1145	...	...	15
Phase #3	1	1300	...	...	35

Figure 4.6: Phase analysis illustration.

Every time when invoked, the Data Analyzer processes all the *analysis intervals* that haven't been analyzed. In our implementation, an analysis interval is one charging cycle, i.e., the time period between two phone charges.

For each analysis interval, eDoctor identifies execution phases by using either RTV or RUV as explained in Section 4.2. To reduce noise and speed up diagnosis, it only records *major phases* - phases that account for more than 5% of the app's total execution time during the last analysis interval. Phases that appear occasionally are likely to be noises.

Each entry in a phase table represents a major phase. Each major phase is identified by a unique phase signature. We use phase signatures to determine which phase a given new resource vector belongs to. For RTV, we use the RTV vector directly as the phase signature; for RUV, we use the combination of center radius of the corresponding cluster as the phase signature (refer to Section 4.2).

For each major phase, Data Analyzer keeps track of its birth timestamp, and its number of appearances and energy consupmtion during each analysis interval. The birth timestamp helps diagnosis by indicating how recently a suspicious phase is first observed. The Diagnosis Engine also uses this information to correlate suspicious phases with triggering events (Section 4.3.3). For the last two variables (appearance count and energy consumed), only the most recent K intervals of data are maintained. Clearly, a large K allows for detection of issues that are introduced earlier, but it incurs larger storage overhead. We find $K = 5$ (about one week in time) strikes a good balance in the trade-off.

4.3.3 Diagnosis Engine

When users notice ABD issues, they invoke the Diagnosis Engine in eDoctor, which pinpoints the culprit app and the causing event. It does so by analyzing all historical phase tables (populated by the Data Analyzer, Section 4.3.2) and event records (collected by the Information Collector, Section 4.3.1), and correlating them to identify the culprits.

Diagnosing Culprit Apps

Identifying the culprit app and the causing event is not trivial. Our approach is based on a key general observation: most ABD issues involve a new, energy-heavy execution phase emerging in a particular app. For example, in the Facebook bug mentioned in Section 1, such new phase is characterized by the wakelock being held for a long time while other resources are used little in the meantime. This phase rarely exhibited before the buggy upgrade of the app. Thus, it is critical for eDoctor to look for such energy-heavy, new phase where ABD is most likely occurring. We refer to such phase as *suspicious new phase* (SN-Phase), and any app that contains an SN-Phase as a *suspicious app*. Essentially, the diagnosis process has two major steps: (1) identifying suspicious apps, and then (2) identifying suspicious causing events.

Step 1: Identifying suspicious apps. eDoctor first prunes out apps that consume low energy, because they are unlikely the root cause of noticeable ABD. In our implementation, Diagnosis Engine only considers the top apps that, combined, consumed 90% of the energy. It then checks whether there is any recent new phase with high energy consumption, i.e., SN-Phase. Determining whether a phase is energy-heavy or not is straightforward (e.g., by computing its energy consumption percentile in the app). But how to define *new*? Users may not start diagnosis immediately after an ABD issue happens. In other words, ABD may start well before the moment of diagnosis. In consideration of this, Diagnosis Engine uses a progressive strategy to search for suspicious apps as follows.

Recall that within an app, each major phase's information is recorded for the K most recent analysis intervals (i.e., charging cycles), which we notate as

τ_1, τ_2, ..., τ_K, where τ_1 is the most recent interval and τ_K is the oldest interval. The Diagnosis Engine first assumes that the noticed ABD originally happened in τ_1. It thus treats those phases with birth timestamps falling in τ_2 to τ_K as normal ones where no ABD occurred. It then checks if τ_1 has any new energy-heavy phase appearing compared to the previous $K - 1$ intervals. If it does not find any, it then assumes the ABD started in τ_2 (and may continue in τ_1), thus it checks whether any SN-Phase exists in the most recent two intervals compared to the previous $K - 2$ intervals. The process goes on until it finally identifies an SN-Phase within the app or it has exhausted all data in the phase table. For apps that are recently installed, they may not have much information in previous intervals. In such cases, any phase that consumes a high level of energy in recent analysis intervals is still considered to be an SN-Phase (when there is no previous intervals to compare). As mentioned before, all apps that contain SN-Phases are then regarded as suspicious apps. Based on our extensive empirical experiments (Section 4.4), there are usually at most 2–3 suspects after this step.

Step 2: Identifying suspicious causing events. For each suspicious app, the event that immediately precedes its SN-Phase is considered the most suspicious in causing the ABD. Diagnosis Engine finds it by comparing the timestamp of the SN-Phase and the timestamps in the event logs.

Finally, the Diagnosis Engine ranks all suspicious apps based on the total energy consumed in their SN-Phase(s). For user convenience, eDoctor reports only the top ranked suspicious app and causing event for repair advisor. Certainly, it could also report all suspicious apps to experienced users if necessary.

Diagnosing System Configurations

Different from per-app resource usages, it is intuitively difficult to find regular patterns on system-wide resource usages (e.g., screen-on time) since they aggregate the usages across different apps. Therefore, Diagnosis Engine does the following diagnosis steps by scanning system-wide configuration event logs backward from the current interval. For each event log, it first identifies the resource type related to the event. For example, for the change of background data setting, it will particularly investigate the "amount of sent/received data" among the usage history of 3 system-wide resources (section 4.3.1). Second, regarding the resource, it finds a common range of the resource usage (in terms of usage time or amount of data transferred) by using an Empirical Rule [81]. Finally, it checks the data after the event to see if most (¿80%) data falls out of the common range found above. If so, the resource highly likely has been abnormally used due to the system-wide configuration, and thus the configuration event is reported to the Repair Advisor.

Suspicious Resource Usage	**Suspicious Event**
CPU	Increased CPU frequency
Display	Increased LCD brightness
Display	Increased timeout
Bluetooth	Toggled Bluetooth on
GPS	Toggled GPS receiver on
Network	Type (2G/3G/4G) upgraded
Network	Toggled Wi-Fi connection on
Network	Toggled Airplane mode off
Network	Toggled the background data setting on
Any	Upgraded system
Any	Switched firmware

Table 4.1: Mapping for pinpointing system wide culprit events.

4.3.4 Repair Advisor

In addition to providing a diagnosis report about the suspicious apps and triggering events, eDoctor also suggests the most suitable repair solutions based on the symptom and triggering events. This section explains how eDoctor determines the repair suggestions, and potential ways of repairing ABD issues automatically without user actions. We leave the implementation and evaluation of automatic repair to future work.

Reverting Problematic Apps

If a recent update contains an ABD issue, eDoctor suggests to revert the problematic app back to the previous version or uninstall the app. Unfortunately, Android does not allow reverting apps directly. A tech-savvy user can revert an app with command line tools if a previous version is accessible. A better solution is to revert apps automatically by backing up prior installation packages. When Android installs an app, it stores the installation package on the phone temporarily, but it keeps only the last installed version of the package. If we back up prior versions, we can allow users to install prior versions. eDoctor has implemented a prototype and proved the feasibility of this approach.

Terminating Apps After Use

If the user wants to keep using the problematic version of the culprit app, eDoctor suggests temporary repair solutions in certain scenarios. One of the most common symptoms of energy bugs is that an app continues to consume resources even after the user stops using the app. In this case, eDoctor suggests users to manually terminate the problematic app every time after closing it, so it will not run in the background. As this can be troublesome, a better solution is to have eDoctor automatically terminate the problematic app.

Reverting Configuration Changes

If a recent configuration change causes an ABD issue, eDoctor presents users the identified configuration entry, together with its current and old values. It relies on the user to revert the configuration back to the old setting. User level apps do not have the permission of directly changing configuration values. However, if implemented in the Android framework, it is possible to automatically fix configuration issues.

4.3.5 Automatic Fixes

Although the Repair Advisor suggests the resolution to users, sometimes it is still difficult for regular users to apply them. In addition to giving user advise, we also implemented several proof-of-concept features in eDoctor to automatically fix some of the issues.

App Terminator

If the root cause is related to a particular app overusing resources in background, eDoctor uses its *App Terminator* to kill the app if the user no longer interacts with it but it is still running in background consuming resource. App Terminator maintains a blacklist of problematic apps diagnosed by eDoctor. Every time after the screen is turned off (i.e., after the user stops interacting with the phone), the App Terminator allows apps in the blacklist to run for a certain amount of time. If they keep consuming resource after that, it will terminates the apps. Apps are automatically removed from the blacklist once they are upgraded, in case a new version has fixed the problem.

App Reverter

If a problematic app start draining battery fast only after recent upgrades, eDoctor can reverting them back to the previous version upon users' permission. To achieve this, eDoctor backs up installation packages of previous versions whenever apps are upgraded. But due to the storage constraint on smartphones, we do not back all versions of all apps. First, only the version immediately prior to the current version is backed up. Second, we do not back up apps that are larger than 20MB. Third, we set a maximum space that is allowed to store backups. When the maximum space is reached, we remove the oldest backup. If there is no available backup of previous versions, eDoctor suggests to uninstall the problematic app.

Environment Adaptor

When eDoctor detects any suspicious environmental changes that may drain battery, it automatically configures the phone to minimize the impact of those changes. (1) When the phone is in a location where the user does not have usable (in terms of connectivity) Wi-Fi access points, eDoctor automatically turns off the Wi-Fi interface (of course, users have the option of manually turning it on if needed). (2) When the radio signal is weak, eDoctor reminds users to turn on Airplane mode. (3) When the phone is in an area where 3G/4G signal is weak, eDoctor automatically switches to 2G to save energy.

Configuration Reverter

If there are any suspicious configuration changes, eDoctor presents users the configuration entry, problematic values and unproblematic values. Users can decide if they want to revert to previous values. For some system configuration entries, eDoctor can automatically change it upon users' permission.

4.4 Evaluation

To assess the effectiveness and performance overhead of eDoctor, We used real-world ABD issues to conduct both in-lab experiments and a controlled user study.

4.4.1 Effectiveness (User Study)

We believed that it is important to evaluate eDoctor on real user phones, where the triggering ABD issue is mixed with other healthy apps and normal uses of the phone. Thus, we recruited 31 Android users via campus-wide mailing lists in two major universities - University of California at San Diego (USA) and Peking University (China). The phones they used consisted of 26 different devices with 11 different Android versions and various configurations and usage patterns.

Since a real user study that asks the user to use eDoctor to troubleshoot a naturally occurred ABD issue may take months and a large number of participates to have sufficient data points, we conducted a more controlled experiment. We emulated real-world scenarios where a user performed an ABD-triggering event (e.g., installing a buggy app or misconfiguring a setting), used the phone for some time, noticed rapid battery drain, and then started diagnosis. The whole study took 7-10 days for each participant.

ABD issues were notoriously hard to reproduce due to their dependency on specific phone hardware/software setup and the unavailability of buggy versions. We finally reproduced 17 *real-world* ABD issues and generated 4 injected issues (Table 4.3) by modifying open-sourced Android apps.

App Name	**Category**	**Description**	**Downloads**
Anki-Android	Education	A flash card app	100K+
BostonBusMap	Travel	Bus tracking in Boston	50K+
Cool Reader*	Book	An eBook reader	1M+
Eyes-Free Shell	Tools	Eyes free access to apps	10K+
Facebook	Social	Official Facebook app	100M+
Gallery	Media	A 3D gallery app	built-in
K9Mail	Communication	An popular email client	1M+
Marine Compass	Tools	A compass app	100K+
MyTracks	Health	Route tracking	5M+
Nice Compass	Tools	A compass app	1K+
NPR News*	News	NPR News client	1M+
OpenGPS Tracker	Travel	Route tracking	100K+
OpenStreetMap	Productivity	OpenStreetMap viewer	5K+
Replica Island*	Game	An Android game	1M+
Standup Timer	Productivity	A timer app	1K+
Talking Dialer	Communication	A dialer app	50K+
Vanilla*	Music	A music player	50K+
Weather Bug	Weather	A weather reporter	10M+
WHERE	Travel	Location discovery	1M+

Table 4.2: Apps used in the evaluation user study of eDoctor.

We selected only popular apps that have a significant number of users, each of which has different characteristics on resource usage: dominant resources and usage patterns (e.g., frequency and time duration of use). Therefore, we believe the data we collected from the user study were relatively diverse and representative. Table 4.2 lists information about the selected apps. The numbers in the "Downloads" column of Table 4.2 indicate the number of downloads of app from Google Play (as of May 2012). To save space, we use"K" to present 1,000 and "M" for 1,000,000. "build-in" means this app is bundled with some phones. To cover a wider spectrum of resources and usage patterns, we injected four real-world ABD bugs into apps in popular categories. They are marked with the "*" symbol.

For ABD issues caused by software bugs, we prepared two versions of a target app: one with a real-world ABD issue and the other without (i.e., either already fixed or not yet defective). We took similar steps with ABD-triggering configuration changes. Next, we randomly assigned each ABD issue to 1–5 participants, giving us 50 cases in total. In each case, we asked the user to follow three steps: (1) Use the given app (normal version) for at least 5 days. Meanwhile, participants should feel free to use their own apps as usual. (2) Switch the app to the other version (defective) or changing the configuration (to be the incorrect one). To make it easy for participants, we made two packages to perform the switch with a single click. (3) Use the app (defective version) for

App Name	Issue Type	Issue Description
Anki-Android	Bug	Resource leak (Accelerometer)
	Config	Frequent widget refreshing
BostonBusMap	Bug	Resource leak (GPS)
	Config	Enable continuous updates
Cool Reader*	Bug	Resource leak (Wakelock)
Eyes-Free Shell	Bug	Resource leak (GPS)
Facebook	Bug	Resource leak (Wakelock)
Gallery	Bug	Resource leak (Accelerometer)
K9Mail	Bug	Too many trials
Marine Compass	Bug	Resource leak (Magnetic field sensor)
MyTracks	Bug	Resource leak (Wakelock and GPS)
Nice Compass	Bug	Resource leak (Magnetic field sensor)
NPR News*	Bug	Resource leak (GPS)
OpenGPS Tracker	Bug	Resource leak (GPS)
	Config	GPS precision
OpenStreetMap	Bug	Resource leak (GPS)
Replica Island*	Bug	Resource leak (Orientation sensor)
Standup Timer	Bug	Resource leak (Orientation sensor)
Talking Dialer	Bug	Resource leak (Accelerometer)
Vanilla*	Bug	Resource leak (Wakelock)
Weather Bug	Config	Frequent update
WHERE	Bug	Resource leak (GPS)

Table 4.3: ABD issues used in the evaluation user study of eDoctor.

some time until noticeable battery drain, then invoke eDoctor to diagnose it.

In total, we collected 6,274 hours of real-world resource usage data, on which we evaluated eDoctor's effectiveness with the diagnosis results. We also measured its energy, storage, and memory overhead based on the collected data from real users.

Similar to other user studies, there is always an issue with representativeness. In our study, the ABD issues, the apps, the participants, the usage patterns, the interference with other apps, etc., are little different from real usage scenarios. The primary difference with a real usage scenario is the occurrence of the ABD issue. In our study, it is not the participants who accidentally triggered ABD issues; instead, we asked them to do it. However, we did not indicate to eDoctor which changes are the causing events, i.e., eDoctor diagnosed these issues independently.

Diagnosis Result

Figure 4.7 shows the effectiveness results. Overall, eDoctor (with RUV) accurately diagnosed 47 of the 50 ABD cases (94%). "Overall Case" shows the diagnosed cases among all 50 ABD cases. "Resource Leak" and "Other" shows breakdown of two types of ABD cases.

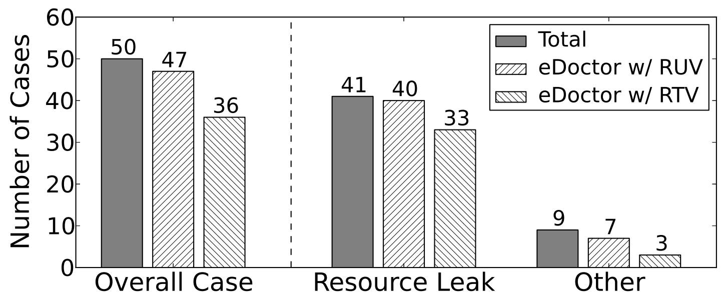


Figure 4.7: Diagnosis results.

eDoctor (with RUV) failed to diagnose 3 cases, although it was able to successfully diagnose the same ABD issues when they took place on some other users' phones. We further analyzed these cases and found that the main reason is that they were caused by more frequent execution of phases that were also common in the past. The first mis-diagnosed case is regarding the Weather Bug app. The user changed a configuration entry to update weather information more frequently in the background. In the mis-diagnosed case, this particular user may have already configured the Weather Bug app to check weather frequently, so changing to the highest update frequency did not introduce a new phase. The second mis-diagnosed case, related to the K9Mail app, is caused by the similar reason.

The third mis-diagnosed case is related to the Vanilla Player app. eDoctor found the app had long period of wakelock-leak like pattern (i.e., wakelock is held but no other resource usage) even when the users was using the version without the bug. It's probably because the user frequently paused the player. In addition, the user did not spend much time using this app, so this wakelock-leak like pattern was common. When the user upgraded to the version that indeed has the wakelock-leak bug, its behavior doesn't show up as a new phase.

In order to reliably diagnose these mis-diagnosed cases, eDoctor needs to conduct more detailed analysis on phases' timing patterns, in addition to their frequency. In other words, even for previously-seen major phases, if the timing patterns change dramatically, it may also indicate abnormal use of resource. We leave it for future work.

RTV vs. RUV. As expected, RUV is more accurate than RTV, where the former diagnosed 47 (94%) while the latter diagnosed only 36 (72%) out of the 50 cases. RUV captures phase characteristics better than RTV, and can detect abnormal phases that use the same resources as their normal counterparts but in abnormal amounts. We also broke down the 50 cases into two high-level categories: resource leaks and other cases. RUV performs better than RTV in both categories. Interestingly, among the two categories RTV is better at resource leaks (80.5%) than others (33.3%). The reason is that resource leaks often involve an app intensively using only one type of resource.

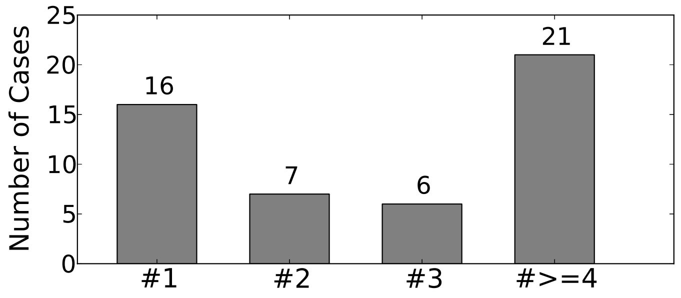


Figure 4.8: Energy consumption rank of the culript app.

Is the culprit app always the biggest energy consumer? As discussed in Section 4.1, one may wonder if existing energy-profiling tools would suffice by simply showing users the top energy consuming app in times of ABD. Our data collected in the user study suggest otherwise. As illustrated in Figure 4.8 [2], only 32% (16) of the cases have a culprit app that ranked #1 in energy use. In almost half (21) of the ABD cases, the rank of the culprit app was actually greater than three. This indicates that existing energy-profiling tools fall short in helping users diagnosing most ABD issues.

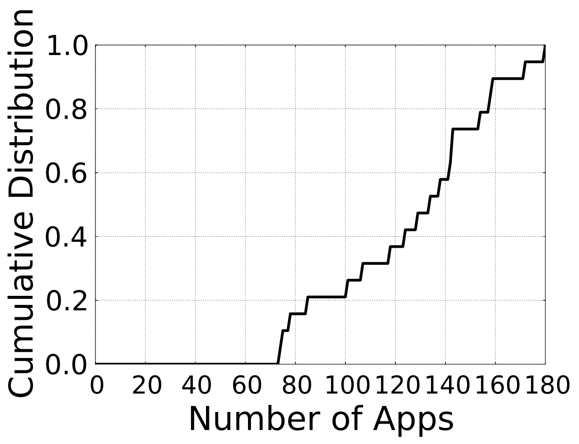


Figure 4.9: Distribution of the number of apps.

How many apps were monitored and how many events happened? One may wonder in the user study how large the pools of apps and events were for eDoctor to identify the culprits. Note that apps that eDoctor needs to monitor may also include background apps/services that the user was not aware of. As

[2]The number at the top indicates the number of ABD cases, e.g., in 21 cases the rank is equal to or greater than 4.

Figure 4.9 shows, for at least 60% of the users, more than 120 apps were run, including those pre-installed apps on the phone.

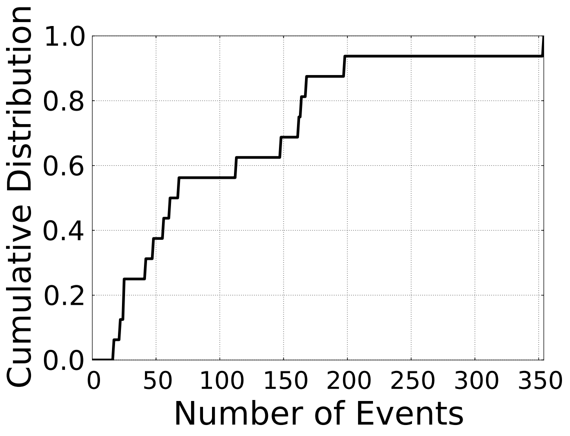


Figure 4.10: Distribution of the number of events.

We also found that many energy-related events happened on the phone during the user study time period (7–10 days). As Figure 4.10 shows, 60% of the users had at least 50 events taking place. This brings a big challenges to users to diagnose, especially many of the events take place without the users' awareness, e.g., automatic app upgrade.

As shown before, eDoctor could diagnose culprits among all these events and monitored apps with high accuracy.

Phase Distribution

To further understand the phase behavior of smartphone apps, we also examined how many normal phases smartphone apps may have. Figure 4.11 shows the cumulative distribution of all 1,890 apps we monitored during the user study. The most important observation is that most apps have a small number of major phases in normal cases. For example, if using RTV (i.e., identifying phases based on resource type), about 80% of the apps have only 1 major phase in normal use and another 13% have only 2. If using RUV (i.e., considering resource usage amount), apps have more major phases but still limited — 80% of the apps have 4 different phases.

Section 4.2 described RUV's normalization method to CPU time. The blue dashed line in Figure 4.11 [3] compares the number of phases with and without normalization. As shown, normalization reduces the number of phases. Nearly 75% of the apps after normalization have only 1 normal phase, making eDoctorś SN-Phase based diagnosis easier.

[3]We only consider the major phases that account for 80% of the total execution time.

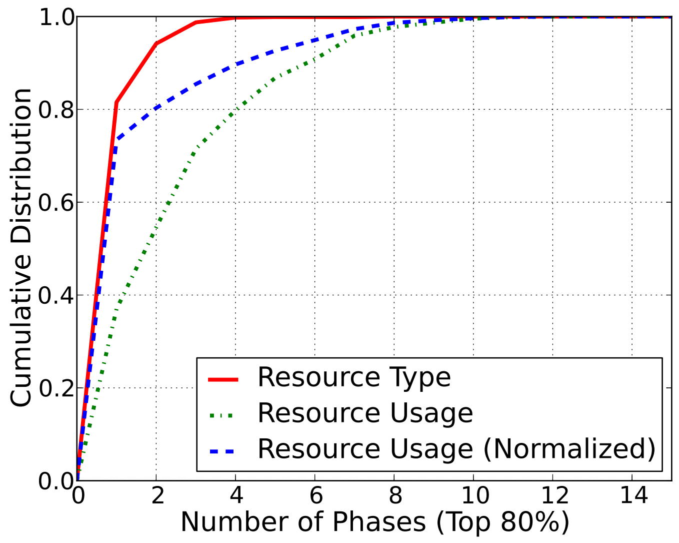


Figure 4.11: The cumulative distribution of number of phases across 1,890 apps we monitored on real user phones during the user study.

4.4.2 Performance Evaluation

We measure the performance of both the online Diagnosis Engine and the offline Data Analyzer, because both of then involve heavy computation.

In online diagnosis, the most time-consuming step is detecting abnormal resource usage in apps. The processing time depends on the number of apps, so we measure the time of analyzing each app. As Figure 4.12 shows, the Diagnosis Engine spends about 30 ms in de-serializing phase table data and about 75 ms in computation for each app. Each bar presents the mean and standard error of the time of diagnosing 25 apps respectively. According to our experiments, eDoctor selects the top suspicious apps (top apps that consume 90% of the energy among all apps) to further analyze, so the app diagnosis can be done within 2 seconds. The total diagnosis is within 2.5 seconds.

During offline data analysis, processing app resource usage data costs most of the time. Processing time is affected by the number of apps and the amount of collected data. Figure 4.13 shows the per-app processing time breakdown by three steps: (a) loading and pre-processing raw data; (b) applying phase analysis and update the phase table; and (c) serializing results. As the data show (Y-axis is in log-scale), step (a) dominates the time due to the large amount of data that need to be processed. The SQLite database engine on Android also has limited performance for large data [99]. This step can be further optimized (e.g., compressing data or ignoring rarely used apps) in the future. We believe that relatively long processing time is not a critical issue because offline analysis is done when the phone is connected to power supply and not being used by the

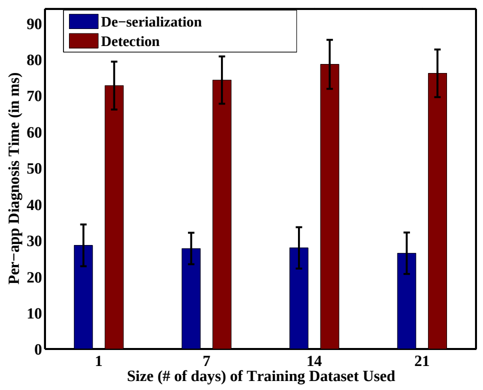


Figure 4.12: Anomaly detection (on-line) time breakdown per app.

user.

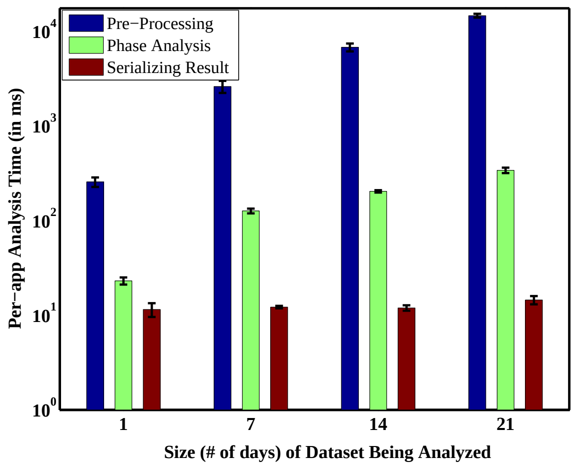


Figure 4.13: Data analysis (off-line) time breakdown per app.

4.4.3 Overhead Evaluation

Since eDoctor's Information Collector runs in the background on a phone from time to time (once per 5 minutes by default when the phone is inactive), the overhead can be a concern. In this section, we report measurements of the energy, storage, and memory overhead of eDoctor on a Nexus One device with 60 apps installed.

Battery Consumption Overhead

We directly measured eDoctor's battery consumption on the Nexus One device. We used a National Instruments NI USB-6210 DAQ to measure the voltage and current on the battery and calculate the power consumption of the entire device.

As shown in Figure 4.14, running eDoctor added only 1.5% power overhead to an *idle* Nexus One (82.5mW) which had no user interaction but only ran built-in system software with Wi-Fi and radio signal enabled. In the figure, baseline (the first three bars): idle Nexus One phone with Wi-Fi and radio signal enabled. eDoctor collects all 60 apps' resource usage on this phone (the fourth bar).

In practice, eDoctor's percentage overhead should be even lower since the user's ordinary use of a phone and additional apps running in the background would wake up the phone. In this case, eDoctor can simply piggyback to collect the resource usage information. The low overhead of eDoctor is not surprising. Instead of monitoring resource usage by itself, eDoctor leverages the low-level resource usage information that is already collected by the Android OS and used for Android's built-in "Battery Usage" utility.

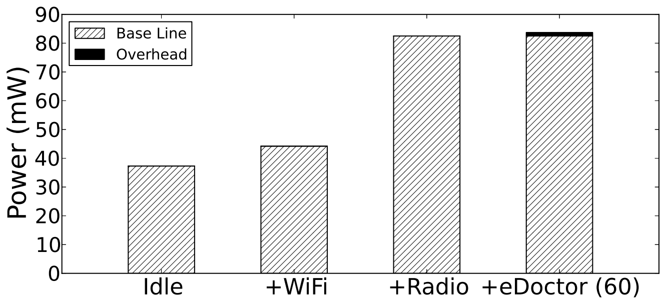


Figure 4.14: eDoctor's battery consumption overhead for data collection.

Storage Overhead

Storage is limited on smartphones, so it is also important for eDoctor to use a small amount of internal flash storage. The main storage consumption by eDoctor is (1) the periodically collected resource data and (2) the table that contains phase information. We measured the storage overhead by running a phone with a default setting of eDoctor for 24 hours. Since the number of apps affects storage overhead, we ran the experiments with 50, 70 and 100 apps installed respectively. Table 4.4 shows that eDoctor used about 2MB per day at most (with even 100 apps). Since eDoctor only keeps information in the past week (configurable), it would use around 15MB storage — an acceptable amount, especially considering that modern smartphones now provide several gigabytes of storage.

Memory Overhead

We used Trepn^{TM} Profiler [43] to measure eDoctor's memory overhead. diagnostic tool that lets you profile the performance and power consumption of Android applications running on devices featuring Qualcomm Snapdragon pro-

Number of apps	**50**	**75**	**100**
Data size (24 hours)	920 KB	1426 KB	1915 KB
Phase information	108 KB	162 KB	216 KB
Total	**1028 KB**	**1588 KB**	**2131 KB**

Table 4.4: Storage used by eDoctor.

cessors. eDoctor's memory footprint was small, 23.3 – 25.2MB, when it ran for data collection. Memory usage did not increase much over time since the data collected were written to flash storage.

4.5 Limitations and Discussions

4.5.1 What Cases eDoctor Cannot Diagnose?

As shown, eDoctor falls short if the abnormal phase where ABD occurs also occurs frequently in the past. However, we find that such cases are relatively rare (e.g., 6%). Another difficult case for eDoctor is that the user happens to install or upgrade two apps at similar times, one normal but energy-savvy, and the other abnormal with an ABD-causing bug. In this case (even though it never happened in our user study with 31 participants for 7–10 days), eDoctor regards both as suspects. Since it reports only the top ranked app to the user, it may result in mis-diagnosis. It may be useful to report both apps, or even better, try to fix one first, and if not resolving ABD, roll back and fix the other. Finally, eDoctor cannot diagnose cases where the causing event happened sufficiently far in the past where eDoctor does not have the event or resource usage information any more (due to storage constraints). In this case, eDoctor could instead flush the history to a remote server.

4.5.2 Is eDoctor Limited to Android?

Although we implemented eDoctor on Android, its approach is not limited to any particular platform. We chose Android because of its openness that allowed us to take advantage of the battery usage information without users having to jailbreak their phones. For other platforms, similar ideas can certainly be implemented by the original platform builder.

4.5.3 Alternative Approaches

While eDoctor leverages the phase behavior of programs to identify abnormal apps, there can be other approaches. For example, one alternative is to analyze the energy consumption history and use signal processing techniques to detect abnormal energy consumption in a way similar to network traffic monitoring for intrusion detection [52]. But such techniques may have large false positives as reported in previous studies, e.g., network monitoring. Another approach is to use a dynamic bug detector to catch battery bugs dynamically. It may have an overhead problem since it needs to monitor at the instruction-level. It may not work for cases caused by unknown bug patterns or misconfiguration.

Using Statistic Approaches Detecting Abnormal Usage

We have also tried several different statistic approaches to detect abnormal usage.

eDoctor periodically collects a set of variables regarding per-app resource usage. A key observation is that when an app behaves normally, its usage of underlying resources likely follows certain patterns. Such patterns can be of two main types: (1) the variable values are mostly within a certain range, and (2) across variables, their values conform to linear relationships. If later any patterns are found to be broken, it is a good indicator of abnormal behaviors. For instance, in the Facebook bug (Table 1.1-a), the variable "periods of time when running in foreground" and the variable "periods of time when holding wakelock" are usually closely correlated, exhibiting an inter-variable pattern. But after the bug is introduced, the pattern breaks because the app holds wakelock even when it is not being actively used. Another example is a "Rhapsody" bug. Since this popular music streaming app needs to download and play music data, normally the ratio between variable "amount of received data" and "when (app) using CPU" is relatively stable. Due to a bug, however, this pattern breaks because the app continues occupying CPU even if the user has turned on Airplane mode (a mode that closes all radio and networking connections).

The goal is to derive patterns about both per-variable ranges and inter-variable relationships from the collected raw data. There are two major challenges: (1) How to accurately capture inter-variable relationship? The analysis technique needs to characterize the most important correlations (if any) among any numbers of variables. Further, this should be achieved without having semantic knowledge about app functions or configurations. (2) How to filter out noise? Apps occasionally use some resources without clear patterns. Such usage does not differentiate normal and abnormal states, thus should be filtered out.

To address these challenges, we use a statistical method known as *principal component analysis (PCA)* [69]. The goal of PCA is to identify the most significant components of variability in high dimensional data. The method works by computing the low dimensional subspace in which the data exhibits the most variance; the basis vectors of this subspace are found from the top eigenvectors of the data's covariance matrix. PCA can be used for anomaly detection by computing the components of each pattern that lie inside and orthogonal to this subspace. If a pattern contains a large orthogonal component, then it is labeled an anomaly.

Here we apply PCA to the resource usage data of *each app*. We use PCA to determine the typical variabilities in such data as occur during normal usage. Once these variabilities are identified, we then monitor subsequent resource usage for significantly different components of variability. When such components are present in the data, it indicates that an app is behaving abnormally.

More specifically, we treat each record of an app's resource usage as a p-dimensional vector $\vec{v} \in \Re^p$. The p elements of these vectors measure the usage of different types of resources during a small time period. We apply PCA to the usage record vectors collected during each app's normal execution *without* battery drain symptoms. We compute the covariance matrix of this data and identify its top k eigenvectors $(\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_k)$ as the k principal components of variability during normal execution.

Online Diagnosis. We detect anomalies in new resource usage vectors by computing the distance that they lie from the subspace spanned by the top k principal components (as derived from Data Analyzer, Section 4.3.2). For a vector $\vec{v}$, this distance is given by $d = \sqrt{\sum_{i=1}^{p} v_i^2 - \sum_{i=1}^{k} (\vec{v} \cdot \vec{e}_i)^2}$ (Eq. 2), where the eigenvectors $\vec{e}_i$ are assumed to be normalized to unit length. We label the resource usage as abnormal if this distance exceeds a particular threshold $d > d_{\text{thresh}}$.

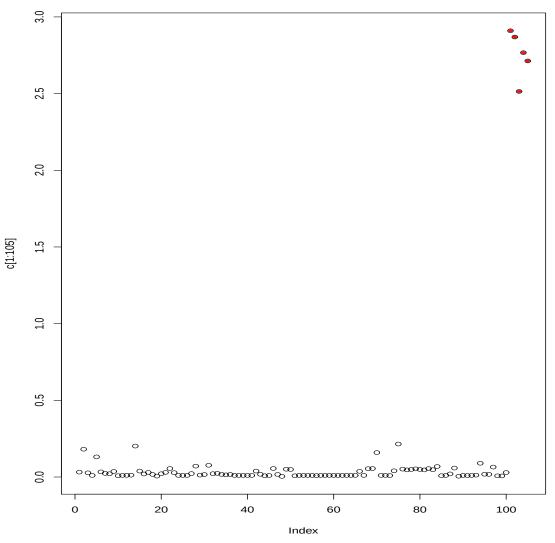


Figure 4.15: Applying PCA-based approach on the bug in the Facebook App.

Figure 4.15 illustrates the idea with the bug in the Facebook App. The white circles are data points collected from a version without the bug, and the red circles are the data points collected from the version with the bug. The Y-axis value is the distance of the resource usage vector from the subspace spanned by the top k principal components. As it shows, these abnormal data points are obviously farthur away from the subspace comparing to the normal ones.

There are two free parameters in the above framework that must be determined heuristically. The first is the number of principal components, k, used to represent the variability during normal usage of each app. We select this value so that the top k principal components capture at least 95% of the data's variance. (Put another way, we choose k such that the top k eigenvalues of the covariance matrix sum to over 95% of its trace.) This heuristic is recommended for many applications of PCA.

The second free parameter in this framework is the threshold d_{thresh}. We choose this threshold heuristically by examining histograms of distances from Eq. 2 during normal and abnormal resource usage. In practice, an effective threshold is one that covers about 90% of the distances observed during normal usage.

We have test PCA-based approaches in some in-lab experiements. It works well for many ABD issues, but it has some inherited disadvantages. When an app has two relatively different usage scenarios, which may have two relatively different multi-variable patrerns among its usage data. This will confused PCA to make wrong decision. It is not uncommon to complex apps like the Facebook app.

5 Related Work

5.1 Energy Consumption Modeling and Measurement

Work has been done to model and measure energy consumption on smartphones, providing guidance on energy-efficient software development.

Shye et al. [98] study mobile architectures in their natural environment – in the hands of the end user. Specifically, it develops a logger application for Android G1 mobile phones and releases the logger into the wild to collect traces of real user activity. It then shows how the traces can be used to characterize power consumption, and guide the development of power optimizations. To quantitivly measure power consumption, it presents a regression-based power estimation model that *only* relies on easily-accessible measurements collected by our logger. The model accurately estimates power consumption and provides insights about the power breakdown among hardware components.

Solid energy management requires a good understanding of where and how the energy is used. Carroll and Gernot [54] present a detailed analysis of the power consumption of a recent mobile phone, the Openmoko Neo Freerunner. They measure not only overall system power, but the exact breakdown of power consumption by the devices main hardware components. They present this power breakdown for micro-benchmarks as well as for a number of realistic usage scenarios. In addition, they also validate these results by two other devices: the HTC Dream and Google Nexus One.

Zhang et al. [104] presents an on-line power estimation and model generation framework. It is designed for developers to have detailed profiling information. The PowerTutor power estimation tool informs smartphone developers of the power consumption implications of decisions about application design and use. The power model in PowerTutor includes six components: CPU and LCD as well as GPS, Wi-Fi, audio, and cellular interfaces. For 10-second intervals, it is accurate to within 0.8% on average with at most 2.5% error. More importantly, this papers makes a pratical contribution - a software implementation of the power estimation tool has been publicly released on the Google Android. Application Market.

Pathak, et al. [84] presents a system-call-based power modeling approach which gracefully captures both utilization-based and nonutilization-based power behavior. The experimental results on Android and Windows Mobile using a diverse set of applications show that the new model drastically improves the accuracy of fine-grained energy estimation as well as wholeapplication energy estimation. We further presented a proof-of-concept demonstration of eprof, the energy-counterpart of gprof, for optimizing the energy usage of application programs. Its power modeling study also exposed significant diversity of power behavior of different OSes and smartphone handsets. As a continous work, Pathak, et al. [83] uses eprof to analyze energy consumption of several popular apps. Eprof sheds lights on internal energy dissipation of these apps and exposes surprising findings like 65%-75% of energy in free apps is spent in third-party advertisement modules. Eprof also reveals several wakelock bugs, a family of energy bugs in smartphone apps, and effectively pinpoints their location in the source code.

Both [104] and [84] are useful for developers to understand how software consumes energy. However, in order to get accurate estimation, these work introduce high overhead to log power usage trace. In contrast, eDoctor has a different goal. Instead of providing detailed and accurate power consumption to developers, eDoctor aims to diagnose why battery drains. It logs resource usage data in a coarse granularity, which is enough to detect abnormal usage and compare relative energy consumption between apps.

5.2 Abnormal Energy Usage Detection

Kim et al. [71] proposes a power-aware malware-detection framework. This work also detects abnormal energy usage, however, there are some major differences: (1) while they model power for detection only, we more sensitively model resource usage, power, and app/system events, so that we can automatically diagnose and resolve the issues; (2) their target is malware, whereas we focus on general ABD issues; such different threat models result in many different design choices; (3) while their evaluation only considers proof-of-concept malware, we have evaluated real-world apps by user study, demonstrating eDoctor's effectiveness and practicability.

5.3 Energy-efficient System Design

A lot of work has been done to build more energy-efficient smartphone system, which cover the wide spectrum of system design, from hardware architecture to applications.

As smartphones require great computation power, processor designers need to explore energy efficiency. GreenDroid [67] is a recenlt work that introduces conservation cores. Conservation cores, or c-cores, are specialized processors that focus on reducing energy and energy-delay instead of increasing performance. Its results show that conservation cores can reduce energy consumption by up to 16.0x for functions and by up to 2.1x for whole applications, while patching can extend the useful lifetime of individual c-cores to match that of conventional processors.

Operating systems on smartphones also require special design and techniques to reserve energy. ECOSystem [103] explores how to support energy as a first-class operating system resource. To traditional operating system, energy, because of its global system nature, presents challenges beyond those of conventional resource management. To meet these challenges ECOSystem proposes the *Currentcy Model* that unifies energy accounting over diverse hardware components and enables fair allocation of available energy among applications. Experimental results show that ECOSystem accurately accounts for the energy consumed by asynchronous device operation, can achieve a target battery lifetime, and proportionally shares the limited energy resource among competing tasks.

Cinder OS [93]) uses techniques similar to existing systems to model device energy use, while going beyond the capabilities of current operating systems by providing an IPC system that fundamentally accounts for resource usage on behalf of principals. It extends this accounting to add subdivision and delegation, using its reserve and tap abstractions.We have described and applied this system to a variety of applications demonstrating, in particular, their ability to partition applications to energy bounds even with complex policies.

Besides re-designing the whole operating system, work has been conducted to make sub-components of an operating system more energy efficient. Anand et al. proposed interfaces that allow apps to actively query device states and issue ghost hints, based on which a middleware layer can support adaptive disk cache management, thus preserving energy [50]. quFiles [100] is a file-system abstraction for representing logical data in different contexts, with which energy consumption can be saved by storage optimizations. Lebeck et al. [75] proposes page allocation schemes to reduce energy consumption and access delay. Cooperative I/O [101] suggests a new I/O interface for apps to defer requests in order to create longer idle period for devices to stay in low-power mode. MAUI [57] automatically off-loads computation to remote servers to save energy on smartphones. Bickford et al. [53] studies the tradeoffs between security versus energy in malware detection.

The LED display on smartphones are also among the most energy consuming hardware components. Anand et al. [48] shows how tone mapping techniques can be used to dynamically increase the image brightness, thus allowing the LCD backlight levels to be reduced. This saves significant power as the majority of the LCDs display power is consumed by its backlight. Its measured analytical results for two different games (Quake III and Planeshift), and user study results (using Quake III and 60 participants) shows that it can save up to 68% of the display power without significantly affecting the perceived gameplay quality.

Dong et al. [60] presents Chameleon, a color-adaptive mobile web browser to reduce the energy consumption by OLED mobile systems. Chameleon is able to reduce the system power consumption of OLED smartphones by over 41% for web browsing, without introducing any user noticeable delay

Wireless networking is also critical to smartphone battery life. STPM [49] is a scheme of self-tuning power management in wireless networks. It adapts to hardware/software environments, and reduces total energy usage of mobile devices. SALSA [92] is an algorithm that automatically adapts to networking channel conditions and requires only local information to decide whether and when to defer large data transmission to save energy. Bartendr [95] schedules cellular data transmission in an energy-efficient fashion based on signal strength prediction. PGTP [47] is an energy-efficient transport protocol for multi-player mobile games. SleepWell [78] achieves energy efficiency by evading Wi-Fi network contention.

Research has also been done to improve the energy efficiency of high level application and services, e.g., location service (Micro-blog [65], EnTracked [72], EnLoc [55] and A-loc [77]).

The above previous work achieves great improvement in smartphone battery usage, yet as discussed before they only focus on *normal* circumstances, i.e., where energy is indeed needed for expected system/app behaviors. In comparison, eDoctor targets at a different yet increasingly important set of problems - abnormal battery drain. It can be noticed that eDoctor also adopts ideas of the above work to minimize its own energy consumption.

5.4 Abnormal Battery Drain Studies

As a result of the paradigm shift in smartphone industry (discussed in Section 1), ABD issues become an emerging research topic that drawn new attention. Pathak et al. [82] also studies characteristics of battery issues on Android system. Our study shares common findings with [82], but there are also many differences. For example, we find very few cases (2.6%) where the battery becomes bad, but [82] finds 15.7% of the cases of this type. We think the reason of contradictory findings are the result of different methodologies. First, [82] counts “post” whereas our study counts “thread”. Counting “post” may not be able to well present the distribution of types of issues, because a post could be a reply, not inquiry of an issue. Second, [82] uses machine learning approaches to cluster “post”, whereas we manually read each “thread” in our study. Machine learning approach is more scalable, but it may also introduce errors. Last but not least, our work applies what learned from the characteristic study to design, implement and evaluate eDoctor, a tool that automatically helps users diagnose and fix ABD issues.

6 Conclusion

6.1 Thesis Achievements

The past few years have witnessed an evolutionary change in the smartphone ecosystem. Smartphones have gone from closed platforms containing only pre-installed applications to open platforms hosting a variety of third-party applications. Unfortunately, this change has also led to a rapid increase in *Abnormal Battery Drain (ABD)* problems that can be caused by software defects, misconfiguration, or environmental changes. Such issues can drain a fully-charged battery within a couple of hours, and can potentially affect a significant number of users.

The main contribution of this thesis is to understand ABD issues and their causes, help smartphone users diagnose ABD issues and assist app developers prevent software bugs that may cause ABD.

- We conducted an empirical study of 537 real-world user-reported battery drain issues sampled from five major smartphone forums (Section 2.1). They covered the two most popular mobile platforms, Android and iOS. We developed a taxonomy for these battery drain issues, and identified their distribtuion in the real world. We found that software problems accounted for a significant portion of the battery drain issues (39.2% on Android, 35.1% on iOS) compared to other root causes.

- We designed and implemented *eDoctor*, a practical tool to help users troubleshoot ABD issues on smartphones. eDoctor runs as a light-weight service on a smartphone to record resource usage and relevant events. It then uses this information to diagnose ABD issues and suggest resolutions. To be practical, eDoctor meets several objectives, including (1) low monitoring overhead (including both performance and battery usage), (2) high diagnosis accuracy and (3) little human involvement. In our user study with 21 ABD issues and 31 participants, eDoctor successfully diagnosed 47 out of 50 cases with only small battery and storage overhead.

- We study 117 battery-related software problems in the Android operating system (with about four years of development history) and 29 popular open source Android apps. From them, we characterized common mistakes programmers make that could lead to battery drain. Based on the results, we provide practical implications for system and app developers.

6.2 Future Work

We plan to release eDoctor on Google Play so that it can help real users while also collect feedback for further improvement. By doing so, we can also extend eDoctor in many ways. For example:

- Collecting phase information from massive amount of users to improve the accuracy of diagnosis. We have found that phase behaviors of a given app are relatively consistant across different users in our user study. However, apps with complicated features may have different usage patterns by different usage scenarios.

- Evaluating energy consumption of apps that have similar features and recommending energy-conservative apps. As we discussed before, energy efficiency is often ignore by developers, because (1) one single app is not an obiviously significant energy consumer even if it is not energy efficient and (2) users' purchasing decisions are largely made on app features and user interfaces. To encourge developers to put more effort on energy efficiency and improve overall battery life for users, we can build an energy efficiency evaluation system based on data from massive users.

More generally, even though eDoctor uses phase behavior and identification to diagnose ABD issues, we believe that it may be useful for other purposes as well, e.g., detecting information leakage, viruses, etc.

We foresee many opportunities to apply static analysis on smartphone apps to prevent energy bugs. In particular, we believe model checking is a suitable technology to analyze smartphone app source code because of its event-driven programming model.

Besides static analysis on source code, profiling application execution is also a promising approach to optimize energy usage and detect energy bugs. The information collected by eDoctor can be directly used by developers to diagnose some of the resource leak bugs.

References

[1] http://developer.android.com/guide/components/services.html.

[2] 4g network drains battery fast. http://goo.gl/4WumI.

[3] Adding more email accounts drains battery. http://goo.gl/Tqyjl.

[4] Android 1.6 platform highlights - battery usage indicator. http://goo.gl/yQwui.

[5] Android central forum. http://www.androidcentral.com/.

[6] Android forum. http://androidforums.com/.

[7] Apple support forum. https://discussions.apple.com/.

[8] comscore reports august 2011 u.s. mobile subscriber market share. http://goo.gl/JTj9C.

[9] Droid forum. http://www.droidforums.net/.

[10] Facebook mobile app stats shocker. http://goo.gl/8HKW7.

[11] Frequent email update drain battery. http://goo.gl/1Ibph.

[12] Gallery sync with picasa albums. http://goo.gl/aSPxC.

[13] Google io 2012 keynote. http://goo.gl/Fo2i0.

[14] Heavy battery drain after gingerbread upgrade on nexus one. http://goo.gl/OkVGi.

[15] ios 5.0.2 is coming next week to solve your iphone 4s battery problems. http://goo.gl/Cf89V.

[16] ios app programming guide - app states and multitasking. http://bit.ly/GGzDpw.

[17] Is facebook 'live feeds sync' causing our battery drain? http://goo.gl/uQmBL.

[18] Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf.

[19] Macrumor forum. http://forums.macrumors.com/.

[20] Number of available android applications. http://goo.gl/Mmuu4.

[21] Our bug reports. http://goo.gl/5c5mb.

[22] Patch: Audiotrack - extend callback thread sleep time. http://goo.gl/sa6AP.

[23] Patch: Enable keeping bt chip in low power mode. http://goo.gl/d1ZtD.

[24] Patch: Fix wakelock leak for interrupted rpc call/reply. http://goo.gl/enKvT.

[25] Patch: Gpslocationprovider eliminate min_fix_count. http://goo.gl/CzyWB.

[26] Patch: Implement smartreply and smartforward for eas. http://goo.gl/LJ7ba.

[27] Patch in android browser: Stop the loading after wakelock timeout. http://goo.gl/kgwcq.

[28] Patch in mytrack: Fixing wakelock being held for too long. http://goo.gl/m3jFx.

[29] Patch: Near-final tweaks to sync timeouts and logging. http://goo.gl/vOGp9.

[30] Patch: Nfc enable low-power rf polling feature. http://goo.gl/3akSO.

[31] Patch: Reduce battery drain caused by insainly high value of widget update. http://goo.gl/dOB8O.

[32] Patch: Release wakelock on ril request send error. http://goo.gl/YvENw.

[33] Patch: Release wakelock when power removed. http://goo.gl/D1uWa.

[34] Patch: Set maximum screen off cpu frequency to 700mhz. http://goo.gl/CAmwQ.

[35] Patch: Shut down the sockets at an earlier point. http://goo.gl/uWfxS.

[36] Patch: Tune notification led blink ramping to reduce power. http://goo.gl/CFxbx.

[37] Patch: Used setinexactrepeating() to fire alarms if appropriate. http://goo.gl/j7lGn.

[38] Patch: Wifi power management change. http://goo.gl/yzuIY.

[39] Remove the tree map in the http headers. http://goo.gl/cxpXI.

[40] Smart phones overtake client pcs in 2011. http://goo.gl/iT86O.

[41] Sprint forum: Htc evo maintenance release - android 2.3 (4.24.651.1). http://goo.gl/SUiPF.

[42] Suffering from 24% per hour drain. http://goo.gl/d1Ajh.

[43] Trepn ^{TM}profiler. http://goo.gl/OLPyM.

[44] Turning on bluetooth drains battery. http://goo.gl/Kcs3q.

[45] Turning on locaton service drains battery. http://goo.gl/tR3r3.

[46] Wikipedia: ios app store. http://goo.gl/iz8ru.

[47] B. Anand, J. Sebastian, S. Ming, A. Ananda, M. Chan, and R. Balan. Pgtp: Power aware game transport protocol for multi-player mobile games. In *2011 International Conference on Communications and Signal Processing (ICCSP)*, pages 399–404.

[48] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 57–70. ACM, 2011.

[49] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. MobiCom '03, pages 176–189. ACM, 2003.

[50] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in the machine: interfaces for better power management. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '04, pages 23–35. ACM, 2004.

[51] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 280–293. ACM, 2009.

[52] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *In Internet Measurement Workshop*, pages 71–82, 2002.

[53] J. Bickford, H. A. Lagar-Cavilla, A. Varshavsky, V. Ganapathy, and L. Iftode. Security versus energy tradeoffs in host-based mobile malware detection. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 225–238. ACM, 2011.

[54] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21. USENIX Association, 2010.

[55] I. Constandache, S. Gaonkar, M. Sayler, R. Choudhury, and L. Cox. EnLoc: Energy-efficient localization for mobile phones. In *INFOCOM 2009, IEEE*, pages 2716–2720, 2009.

[56] R. Cozza, C. Milanesi, A. Gupta, H. J. D. L. Vergne, A. Zimmermann, C. Lu, A. Sato, and T. H. Nguyen. Competitive landscape: Mobile devices, worldwide, 3q10. *Gartner Research Report*, 2010.

[57] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62. ACM, 2010.

[58] CyanogenMod. Cyanogenmod commit: Get the wakelock only if it isn't held already. `http://goo.gl/8veZf`.

[59] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. MICRO 36, pages 217–, Washington, DC, USA, 2003. IEEE Computer Society.

[60] M. Dong and L. Zhong. Chameleon: a color-adaptive web browser for mobile oled displays. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 85–98. ACM, 2011.

[61] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[62] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. PACT '03, pages 220–, Washington, DC, USA, 2003. IEEE Computer Society.

[63] D. Freedman, R. Pisani, and R. Purves. *Statistics, 3rd Edition.* W. W. Norton & Company., 1997.

[64] D. Freedman, R. Pisani, and R. Purves. *Statistics, 3rd Edition.* W. W. Norton & Company., 1997.

[65] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt. Microblog: sharing and querying content through mobile phones and social participation. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 174–186. ACM, 2008.

[66] Google. Android activity lifecycle. `http://goo.gl/5xvtY`.

[67] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P.-C. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31:86–95, March 2011.

[68] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. ISCA '03, pages 157–168, New York, NY, USA, 2003. ACM.

[69] I. T. Jolliffe. *Principal Component Analysis.* Springer, second edition, Oct. 2002.

[70] K9Mail. K9mail commit: Fixed battery drain and delete messages. `http://goo.gl/yIXpV`.

[71] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 239–252. ACM, 2008.

[72] M. B. Kjǽgaard, J. Langdal, T. Godsk, and T. Toftkjǽr. Entracked: energy-efficient robust position tracking for mobile devices. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, MobiSys '09, pages 221–234. ACM, 2009.

[73] Lateroid. Juicedefender, an battery saving application on android. `http://latedroid.com/juicedefender`.

[74] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ISPASS '05, pages 135–146, Washington, DC, USA, 2005. IEEE Computer Society.

[75] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *SIGARCH Comput. Archit. News*, 28:105–116.

[76] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: using low-power processors in smartphones without knowing them. ASPLOS '12, pages 13–24, New York, NY, USA, 2012. ACM.

[77] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao. Energy-accuracy trade-off for continuous mobile device location. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 285–298. ACM, 2010.

[78] J. Manweiler and R. Roy Choudhury. Avoiding the rush hours: Wifi energy management via traffic isolation. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 253–266. ACM, 2011.

[79] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[80] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software, 2007.

[81] M. Owens. *The Definitive Guide to SQLite*. Apress, 2003.

[82] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. HotNets '11. ACM, 2011.

[83] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *EuroSys*, pages 29–42, 2012.

[84] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 153–168. ACM, 2011.

[85] A. Pathak, A. Jindal, Y. C. Hu, and S. Midkiff. Characterizing and detecting nosleep energy bugs in smartphones apps. In *Mobisys*, 2012.

[86] A. Pathak, A. Jindal, Y. C. Hu, and S. Midkiff. Characterizing and detecting nosleep energy bugs in smartphones apps. In *ECE Technical Reports, Purdue University*, 2012.

[87] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[88] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. PACT '03, pages 244–, Washington, DC, USA, 2003. IEEE Computer Society.

[89] G. Perrucci, F. Fitzek, G. Sasso, W. Kellerer, and J. Widmer. On the impact of 2g and 3g network usage for mobile phones' battery life. *Wireless Conference*, pages 255–259, 2009.

[90] R. Powers. Batteries for low power electronics. *Proc. of the IEEE*, 83(4):687–693, apr 1995.

[91] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 321–334, New York, NY, USA, 2011. ACM.

[92] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 255–270. ACM, 2010.

[93] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 139–152. ACM, 2011.

[94] Samsung. Linux kernel shg-i777 commit: Fix fuel alert wakelocks. `http://goo.gl/Kzu8e`.

[95] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. N. Padmanabhan. Bartendr: a practical approach to energy-aware cellular data scheduling. MobiCom '10, pages 85–96. ACM, 2010.

[96] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. ASPLOS-X, pages 45–57, New York, NY, USA, 2002. ACM.

[97] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. ISCA '03, pages 336–349, New York, NY, USA, 2003. ACM.

[98] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. MICRO 42, pages 168–178. ACM, 2009.

[99] R. C. Sprinthall. *Basic Statistical Analysis: Seventh Edition.* Pearson Education Group, 2006.

[100] K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble. qufiles: The right file at the right time. *Trans. Storage*, 6:12:1–12:28, September 2010.

[101] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: a novel i/o semantics for energy-aware applications. In *Proceedings of the 1st conference on Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 117–129. ACM, 2002.

[102] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, Nov. 2006.

[103] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGOPS Oper. Syst. Rev.*, 36:123–132, October 2002.

[104] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114. ACM, 2010.