eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones

Xiao Ma[*†], *Peng Huang*[*], *Xinxin Jin*[*], *Pei Wang*[‡], *Soyeon Park*[*], *Dongcai Shen*[*]
Yuanyuan Zhou[*], *Lawrence K. Saul*[*] *and Geoffrey M. Voelker*[*]
[*]*Univ. of California at San Diego,* [†]*Univ. of Illinois at Urbana-Champaign,* [‡]*Peking Univ., China*

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 or misconfiguration. Such issues can drain a fully-charged battery within a couple of hours, and can potentially affect a significant number of users.

This paper presents 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 no more than 1.5% of power overhead.

1 Introduction

Smartphones have become pervasive. Canalys reported [12] 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 with 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 [40]. 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 measurement [8, 13, 39, 46], modeling and profiling [18, 36, 46, 52], energy efficient hardware [21, 30], operating systems [7, 10, 15, 29, 42, 49, 50, 51], location services [14, 20, 26, 31], displays [5, 17] and networking [4, 6, 32, 41, 43]. 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 work, we address an under-explored, yet emerging type of battery problem on smartphones – *Abnormal Battery Drain (ABD)*.

1.1 Abnormal Battery Drain Issues

ABD refers to 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 batteries might unexpectedly exhaust within hours.

ABD has become a real, emerging problem. When we randomly sampled 213 real world battery issues from popular Android forums, we found that 92.4% of them were revealed to be ABD, while only 7.6% were due to normal, heavier usage (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-a) had a bug that prevented the phone from entering the sleep mode, thus draining the battery in as rapidly as 2.5 hours. The estimated number of its users was more than *12 million* at that time [24], among whom a large portion were likely to have been affected by this "battery bug".

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 (Google Play: 600,000 apps and 20 billion downloads [47]; App Store (iOS): 650,000 apps and 30 billion downloads [2]), however, most app developers are not battery-cautious.

ID	Category	App/System	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" after the user uses 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 to less frequent updates.
(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	The user's office building contains several radiology devices, which interfere with cell signals and thus make the phone spend more power searching signals.	Turn on Airplane mode when the user is in the office.

Table 1: **Representative ABD examples collected from Android forums.**

Smartphone apps used to be primarily made by device manufacturers, with appropriate training and development resources. In contrast, smartphone apps are now mostly developed by third-party or individual developers. They tend to focus limited resources on app features, on which purchase decisions are often made, but put less effort on energy conservation.

(2) The hardware/software configurations and external environments of smartphones have become diverse, making it difficult and expensive to test battery usage under all circumstances. As a result, many battery-related software bugs escape testing, even by professional software teams, e.g., a bug in Android that affected certain Nexus One phones, (Table 1-e), and a bug in iOS that caused a coninuous loop when sychronizing recurring calendar events [11].

(3) In addition to software defects (e.g., Table 1–a, b, d and e), ABD issues can also be caused by configuration changes (e.g., Table 1–c, f) or environmental conditions (e.g., Table 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.

1.2 Are Existing Tools Sufficient?

Existing energy profilers, such as Android's "Battery Usage" utility, PowerTutor [52], and Eprof [36, 35], monitor energy consumption on smartphones. While they provide some level of assistance to developers or tech-savvy users in troubleshooting ABD issues, they are insufficient for broadly addressing ABD issues due to three main reasons:

(1) These tools *cannot differentiate normal and abnormal energy consumption*. A high energy consuming app does not necessarily cause ABD. To determine an app is "normal" or "abnormal", a user needs to know how much battery the app is supposed to consume, which is difficult for typical users, especially since an app's battery usage can fluctuate even with normal usage.

(2) The information provided by these tools requires technical background to understand and take actions on. Even for tech-savvy users, information form these tools are not sufficient for identifying the *ABD causing event* (e.g., an app upgrade). Knowing causing events is critical for pinpointing the right root cause and determining the best resolution.

(3) 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.

Apps like JuiceDefender [27] automatically make configuration changes to extend battery life. They help preserve energy during normal usage, but they cannot prevent or troubleshoot ABD issues.

From a user's point of view, a highly desirable solution is to have the smartphone itself troubleshoot ABD issues 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.

1.3 Our Contribution

This paper presents *eDoctor*, a practical tool to help troubleshoot ABD issues on smartphones. eDoctor records re-

source usage and relevant events, and 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.

To identify abnormal app behavior, eDoctor borrows a concept called "phases" from previous work in the architecture community for reducing hardware simulation time [44, 45]. eDoctor uses phases to capture apps' time-varying behaviors. It then identifies suspicious apps that have significant phase behavior changes. eDoctor also records events such as app installation and upgrades, configuration changes, etc. It 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 controlled user study and in-lab experiments: **(1) User study**: we solicited 31 Android device users with various configurations and usage patterns. We installed eDoctor and popular Android apps with *real-world* ABD issues on their own smartphones. eDoctor could successfully diagnose 47 out of 50 cases (94%). **(2) In-lab experiments**: we also measured the overhead of eDoctor in terms of its energy consumption, storage consumption and memory footprint. The results show that eDoctor adds little memory overhead, and only 1.24 mW of additional power drain (representing 1.5% of the baseline power draw of an idle phone).

2 Real-world Battery Drain Issues

To understand battery drain issues on smartphones, we randomly sampled 213 real-world battery drain issues from three major Android forums: AndroidCentral.com, AndroidForums.com, and DroidForums.net. To effectively sample the issues from the thousands of battery-related discussion threads in each forum, we searched a set of keywords including "battery", "energy", "drain", and their synonyms, and then randomly picked 213 issues that were confirmed to be resolved. With the collected issues, we studied their root cause categories, triggering events and repair solutions found by users (e.g., removing an app or adjusting configuration) to get guidelines for eDoctor's design.

2.1 Root Cause Categories

We studied the root cause categories and distribution of the problematic components (Figure 1). We made the following observations.

(1) *The majority (92.4%) of the sampled battery life complaints by users are related to abnormal battery drain, and only 7.6% are about heavy yet normal battery usage of some mobile apps*. This breakdown indicates that (i) ABD is an emergent and pervasive problem for smartphones, and (ii) before trouble-shooting a battery issue, one may first need to know whether it is indeed caused by some abnormal problems or if it is simply due to heavy usage of the device or a particular app. An energy profiler can give the battery usage of each app, but cannot usually tell whether the usage is normal or abnormal.

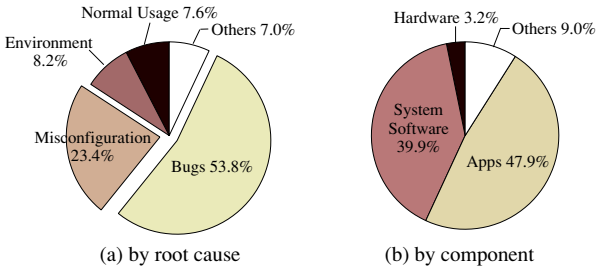


Figure 1: **Distribution of 213 real-world battery drain issues that we randomly sampled**. The meaning of *Others* in each graph: (a) problems with uncommon root causes such as battery indicator error; (b) other sources causing battery drains such as environmental conditions.

(2) *Application issues cause 47.9% of all examined cases*. This observation supports our assertion that app developers are not energy cautious. About three-quarters of the app issues have been identified as app bugs and the remaining are related to configuration. Besides app issues, other factors such as bugs in the system (22.2%), configuration changes (11.8%) and environmental conditions (8.2%) can all lead to ABD issues. It would save a user (even a tech-savvy user) time and effort if a tool can automatically pinpoint the reason for ABD issues and suggest a repair solution accordingly.

(3) *Overusing or misusing certain types of resources can cause ABD issues*. Software bugs and misconfiguration can result in misusing or overusing *certain types of resources*, such as GPS, sensors, etc., leading to an ABD problem. These situations imply that it is beneficial to monitor and analyze usage on those resources. By doing so, eDoctor can separate abnormal from normal battery drains and also suggest detailed repair solutions directly related those resources.

For many ABD issues, especially those caused by misconfiguration and system bugs, it is difficult for an energy profiler to diagnose. For example, enabling background data transmission may result in high energy consumption of certain apps that transfer data when running in the background. Profilers may list these apps as top energy consumer, which mislead users to think they became abnormal and thus remove them.

Events	Cases	Appropriate Solution
App Installation	27	Remove app
App Upgrade	11	Revert to the previous version
System Upgrade	28	Wait for new update
Configuration change	15	Adjust configuration
Environmental change	12	Adjust configuration
Others	16	Others
Not remembered	104	-

Table 2: **ABD-triggering events and the most appropriate resolving solutions.** "Not remembered" refers to the cases where users do not remember what they have done that could possibly cause ABD issues.

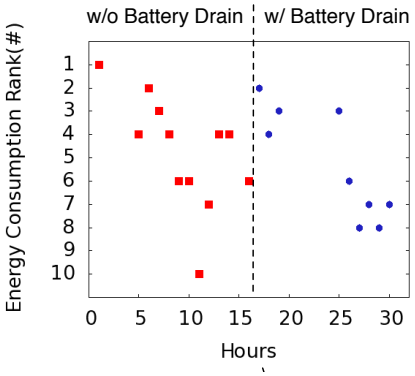


Figure 2: **Battery consumption rank of the Android Gallery app running on a real user's phone**. We recorded the battery consumption rank of this app reported by the Android "Battery Usage" utility, once every hour. The first 15 hours is the time period when the app does not have the battery bug, whereas the second 15 hours is the period when the bug manifested.

2.2 Triggering Events and Resolutions

In general, ABD issues happen only after certain events, 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. Therefore, knowing such triggering events is critical for suggesting appropriate repair solutions to users, as shown in Table 2.

Interestingly, however, in more than 48% of the 213 ABD issues, users did not remember what they had done previously or what could be the possible ABD-triggering event. In such cases, manually diagnosing and resolving the issue becomes difficult. Simply removing a suspicious app, probably the one reported by energy profiling tools as a high energy consumer, is not always the most appropriate solution; it can be either overkill or even incorrect.

3 Execution Phases in Smartphone Apps

To identify the problematic app or system for an ABD issue, it is critical 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 2 from a real case, such approach does not always work because an app's rank in the battery consumption report can fluctuate over time. The challenge is that there is no clear difference between normal and abnormal periods. Thus, energy profiles and rank are not reliable indicators for troubleshooting ABD issues. Additionally, Figure 2 shows that *changes* in battery consumption or rank of an app are also not accurate indicators for abnormal behaviors for similar reasons.

To identify abnormal app behaviors, eDoctor borrows a concept called "phases" from previous work for reducing hardware simulation time [16, 19, 23, 28, 38, 44, 45]. The previous work has shown that 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 [45].

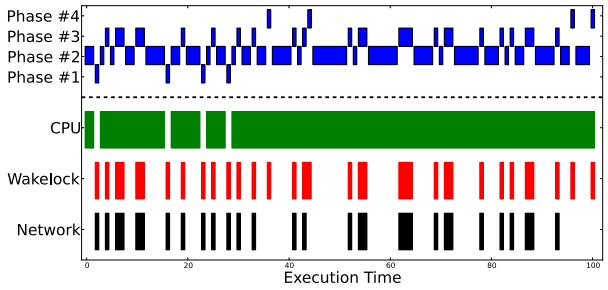
Phase Identification. Inspired by the previous work, eDoctor uses phases to capture an app's behavior in terms of resource usage. The execution of an app is divided into execution intervals, which are then grouped into phases. Intervals in the same phase share similar resource usage patterns. When an 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 relevant events, such as a configuration change, eDoctor can identify both the culprit app and triggering event with high accuracy.

Prior hardware simulation work studied architecture related behaviors (e.g., cache miss ratio), so they captured phases based on instruction-level information, such as basic block vector (BBV). However, such fine-grained information is not suitable for identifying resource usage phases because it does not directly correlate to resource usage. Smartphone apps are different from most desktop or 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 ABD issues. Therefore, we can identify phases by observing how these resources are used by an app during different execution intervals.

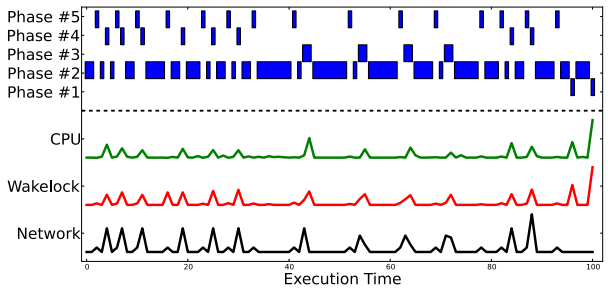
Our first approach starts from a fairly coarse-grain level by recording only resource types used during each execution interval. We refer to this method as *Resource Type Vector (RTV)*. It is based on a simple rationale that different execution phases use different resources. For example,

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 an execution interval. Each bit indicates whether a certain resource type is used in this interval. If two intervals have the same RTV, they belong to the same phase.

As shown in Figure 3(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 frequent 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.



(a) Phase pattern based on RTV



(b) Phase pattern based on RUV

Figure 3: **The phase behavior of the Facebook App in a real user's smartphone**. In the top part of both figures, the shaded bars indicate which phase the app is in. In the bottom part of figure (a), shaded bars indicate the resource is in use. In the bottom part of figure (b), the curves indicate the amount of resource usage.

Although the RTV scheme is simple, it turns out to be too coarse-grained. 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. Therefore, we explored a second scheme — *Resource Usage Vector (RUV)*. Each element in a RUV is the amount of usage of the corresponding resource.

We calculate the usage of a resource by the amount of the resource normalized by the CPU time. The execution interval cannot be too small in order to control the measurement overhead, so an app may run for only a fraction of one execution 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 that belong to the same phase, even if the app runs for different amounts of time in each interval.

If two execution intervals have similar RUVs, they belong to the same phase. Similar to previous work [45], 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. For each k, we evaluate the quality of the clusters by calculating 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 3(b) shows the RUV phase behavior using the same data. As it shows, RUV captures one more phase compared to the phases divided by RTV, enabling eDoctor to further differentiate between low and high network usage. More specifically, phase #3 and phase #4 both have usage of CPU, wakelock and network, but phase #4 has higher network usage. It provides more fine-grained information regarding an application's phase behavior.

4 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 in source code, eDoctor's diagnosis focuses on identifying (1) which app causes an ABD issue and (2) which event is responsible, e.g., the user updated an app to a buggy version or made an improper configuration change. Based on such diagnosis result, eDoctor then suggests appropriate repair solutions.

There are two major challenges involved in achieving these objectives. First, it is non-trivial to accurately pinpoint which app and 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 changes. Second, eDoctor itself should not incur high battery overhead. It needs to balance the energy overhead and the amount of information needed for accurate diagnosis.

This section presents our design of eDoctor. As an overview, eDoctor consists of four major components: Information Collector, Data Analyzer, Diagnosis Engine, and Repair Advisor. The Information Collector runs as a light weight service to collect resource usage and event logs. The Data Analyzer performs phase analysis (Section 3) on the raw data and stores intermediate results to facilitate future diagnosis. Off-line analysis is done only when the phone is idle and connected to external power, in order to avoid affecting normal usage. When users notice ABD, they initiate the Diagnosis Engine to find the culprit app and the causing event. Based on the diagnosis result, the Repair Advisor provides the most relevant repair suggestions.

eDoctor can be installed as a standalone app. It runs on most Android phones and it is compatible with all Android versions since 2.1. A modified Android ROM is optional to track app-specific configuration changes.

4.1 Information Collector

The Information Collector records three main types of 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.

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.

What resource usage information to store depends on the phase identification method (Section 3). RTV uses a bit vector to record whether the resources have been used in each recording interval. 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 resource usage tracking mechanism in the Android framework. This mechanism keeps a set of data structures in memory to track resource usage of each app. The resource usage data are maintained for each individual app, even if multiple apps run during the same recording interval. The values recorded are accumulated amounts since the last time the phone was 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 shows a simplified example of a resource usage table for an app.

Some resources can be simultaneously accessed by multiple apps without consuming extra energy. For example, once a GPS unit is turned on, it gathers location examples, and it does not consume extra energy if more than one app requests those examples. eDoctor performs coarse-grained accounting of such resources; so if N apps access such a resource for overlapped T time units, each app is charged for T time units of resource utilization. Fine-grained energy profilers like Eprof [35] use a proportional accounting scheme, such that each app would only be charged for T/N units of resource utilization. eDoctor's uses the coarse-grained schema because its goal is to track app-specific energy patterns, not overall energy fluctuations of the whole system.

Energy consumption. In addition to resource usage, eDoctor also records battery consumption of each app in each recording interval. Energy consumption is used 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 consumed energy of each app. 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, because it doesn't have to track fine-grained information such as energy state switches. Therefore, we employ an efficient profile-based energy model instead of expensive state-based energy models [46, 52].

Each Android device comes with data about power consumption of various hardware components measured by the manufacture, e.g., the average power consumption of the processor running at different frequencies and the average power consumption of the Wi-Fi device being idle or sending data. eDoctor combines this average power consumption data by the usage data it collects to estimate the total energy consumption of an app during each recording interval. This energy model has been used in both industry (e.g., Android's "Battery Usage" utility [1]) and academic research (e.g., ECOSystem [51]).

Events. Events are critical for both diagnosis and repair advisory. eDoctor records two types of events: (1) configuration changes, and (2) maintenance events (installation, updates). 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. Since most apps use Android's facility components (e.g., `SharedPreferences`) to manage configurations, we track app configurations by modifying these common components. `SharedPreferences` is a general framework that allows developers to save and retrieve persistent key-value pairs of primitive data types, which is suit-

able for managing user preferences. We modified the implementation of the `SharedPreferences.Editor` interface to let it send a broadcast message to eDoctor whenever a preference entry is changed. Each message contains the name of the app, the preference file name, the preference key name and its new value. These messages are identified with a special key and only eDoctor can receive them, so they are effectively unicast messages to eDoctor. One drawback of this approach is that if the developers implement their own mechanisms to manage preferences, eDoctor cannot track the changes. This is rare, however.

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 the system, and switching firmware. eDoctor records these 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`.

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.

4.2 Data Analyzer

eDoctor's Data Analyzer is responsible for parsing all resource usage data collected by Information Collector, generating phase information (Section 3) for each app, and storing it in a per-app *phase table*. 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.

Every time when invoked, the Data Analyzer processes all the *analysis interval*s 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 3. 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 noise.

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 center and the radius of the corresponding cluster as the phase signature (refer to Section 3).

For each major phase, the Data Analyzer keeps track of its birth timestamp, and its number of appearances and energy consumption 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). 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 and computing overhead and potential mis-diagnosis. We find $K = 7$ (about one week in time) strikes a good balance in the trade-off.

Figure 4 illustrates a simplified version of phase analysis. Based on k-means clustering computation (Section 3), entries with timestamp 5, 10 and 25 belong to the same phase (Phase #1 in the Phase Table below), because they have similar normalized 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 for other resources 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 developer forgets to release wakelock.

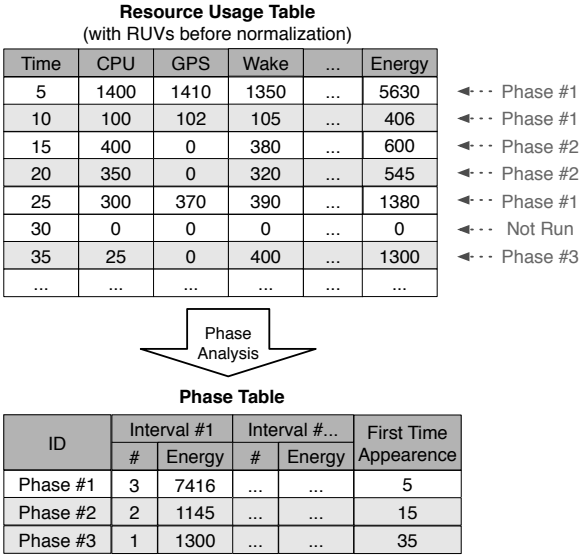


Figure 4: **Phase analysis illustration**. The resource usage table shows seven resource usage records collected by using the RUV method (before normalizing to CPU time).

4.3 Diagnosis Engine

When users notice ABD issues, they invoke eDoctor's Diagnosis Engine, which pinpoints the culprit app and the causing event. It analyzes historical phase tables (calculated by the Data Analyzer, Section 4.2) and event records (collected by the Information Collector, Section 4.1), and correlating them to identify the culprits.

Identifying the culprit app and the causing event is not trivial. As demonstrated in Section 3, energy profile and rank are not reliable indicators for diagnosing ABD issues. Some ABD issues are caused by intensive consumption of certain resources, e.g., GPS or the processor. However, the mere fact that a resource is used for a long time does not necessarily indicate abnormal behavior — an app may simply be designed to run for a long time. In addition, considering only recent resource usage is insufficient, since historical baseline data is needed to identify abnormal behavior.

eDoctor's approach is based on a key 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, a new such 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. We refer to such a phase as a *suspicious new phase* (SN-Phase), and any app that contains an SN-Phase as a *suspicious app*. 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. We only consider the top apps that, combined, consumed 90% of the energy. eDoctor then checks whether there is any recent 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, τ_1 and τ_2, compared to the previous $K-2$ intervals. The process goes on until it finally identifies an SN-Phase or it has exhausted all collected 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 5), there are usually at most 2–3 suspects after this step.

eDoctor keeps a week of historical data for each app, for two main reasons. First, if a new phase appears but the user has been using the app for a week without observing battery issues, that new phase is likely to be legitimate. Second, storing less data helps control the storage overhead and computation time of the data analysis (Section 4.2).

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. The 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.

4.4 Repair Advisor

In addition to providing a diagnosis report about the suspicious apps and causing events, eDoctor also suggests the most suitable repair solutions based on the symptom and causing events.

Uninstalling or reverting a problematic app to a previous version. 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 permission to directly change configuration values. However, if implemented in the Android framework, it is possible to automatically fix configuration issues.

We leave the implementation and evaluation of automatic repair to future work.

5 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.

5.1 Effectiveness (User Study)

We wanted to evaluate eDoctor on real user phones, where ABD issues were mixed with normal usage of phones and apps. 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). These users had 26 different devices with 11 different Android versions and various configurations and usage patterns.

A real user study would ask participants to troubleshoot naturally occurring ABD issues. However, such a study might take several months and require a large number of participants to generate sufficient data points. Thus, 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 hard to reproduce due to their dependency on specific versions of hardware and software. We finally reproduced 17 *real-world* ABD issues. We also generated 4 synthetic issues by modifying open-source Android apps (Table 3). We selected only popular apps that had a significant number of users; these apps also had heterogeneous patterns of user interactivity and resource utilization. Thus, we believe that our user study provides a relatively diverse and realistic sample.

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 from Table 3 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 use their own apps as usual. (2) Switch the app to the defective version, or change the configuration to the incorrect one. To make it easy for participants to do this, we designed custom software that performed the switch with a single click. (3) Use the defective app until ABD is apparent, and then invoke eDoctor to diagnose the problem. In total, we collected 6,274 hours of real-world resource usage data. We used this data to evaluate eDoctor's diagnostic effectiveness, as well as its energy, storage, and memory overhead.

5.1.1 Diagnosis Result

Figure 5 shows eDoctor's effectiveness. Overall, eDoctor with RUV accurately diagnosed 47 of the 50 cases (94% accuracy). eDoctor misdiagnosed three cases. These three ABD issues were experienced by multiple users. eDoctor misdiagnosed these issues for some participants, but successfully diagnosed the issues for other users.

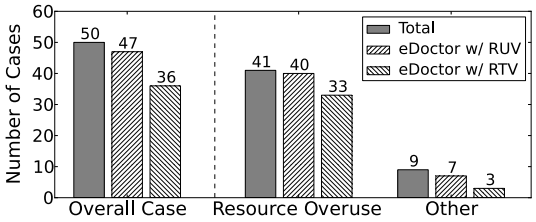


Figure 5: **Diagnosis results**. "Overall Case" shows the diagnosed cases among all 50 ABD cases. "Resource Overuse" and "Other" show breakdown of two types of ABD cases.

There were two reasons for misdiagnosis. First, some ABD issues occurred without an obvious change in the app's phase behavior. For example, at initialization time, one user configured the "Weather Bug" app to frequently update its weather data. High-frequency updates cause ABD, but for this user, the "Weather Bug" app *started* in the ABD state, so eDoctor could not detect anomalies in the app's behavior after the user upgraded to the defective version. eDoctor misdiagnosed ABD with the "K9Mail" app for a similar reason.

eDoctor can also misdiagnose ABD if it lacks sufficient longitudinal data for an ABD-causing app. For example, one user ran the non-buggy version of the "Vanilla

App Name	Category	Description	Downloads	Issue Type	Issue Description
Anki-Android	Education	A flash card app	100K+	Bug Config	Resource overuse (Accelerometer) Frequent widget refreshing
BostonBusMap	Travel	Bus tracking in Boston	50K+	Bug Config	Resource overuse (GPS) Enable continuous updates
Cool Reader*	Book	An eBook reader	1M+	Bug	Resource overuse (Wakelock)
Eyes-Free Shell	Tools	Eyes free access to apps	10K+	Bug	Resource overuse (GPS)
Facebook	Social	Official Facebook app	100M+	Bug	Resource overuse (Wakelock)
Gallery	Media	A 3D gallery app	built-in	Bug	Resource overuse (Accelerometer)
K9Mail	Communication	An popular email client	1M+	Bug	Too many trials
Marine Compass	Tools	A compass app	100K+	Bug	Resource overuse (Magnetic field sensor)
MyTracks	Health	Route tracking	5M+	Bug	Resource overuse (Wakelock and GPS)
Nice Compass	Tools	A compass app	1K+	Bug	Resource overuse (Magnetic field sensor)
NPR News*	News	NPR News client	1M+	Bug	Resource overuse (GPS)
OpenGPS Tracker	Travel	Route tracking	100K+	Bug Config	Resource overuse (GPS) GPS precision
OpenStreetMap	Productivity	OpenStreetMap viewer	5K+	Bug	Resource overuse (GPS)
Replica Island*	Game	An Android game	1M+	Bug	Resource overuse (Orientation sensor)
Standup Timer	Productivity	A timer app	1K+	Bug	Resource overuse (Orientation sensor)
Talking Dialer	Communication	A dialer app	50K+	Bug	Resource overuse (Accelerometer)
Vanilla*	Music	A music player	50K+	Bug	Resource overuse (Wakelock)
Weather Bug	Weather	A weather reporter	10M+	Config	Frequent update
WHERE	Travel	Location discovery	1M+	Bug	Resource overuse (GPS)

Table 3: **Apps and ABD issues used in our experiments**. The numbers in the "Downloads" column indicate the number of app downloads from Google Play, as of May 2012. To save space, we use "K" to present 1,000 and "M" for 1,000,000. "Built-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. "Resource overuse" indicates a bug that uses a resource for longer than necessary, e.g., the developer forgets to release a resource after using it or holds a resource for too long.

Player" app for a short amount of time. During this short time period, the app displayed behaviors that resembled a wakelock leak (this might have occurred because the user frequently paused the player). The user soon updated to the defective version of the player that *did* have a wakelock leak. However, eDoctor did not detect a new phase, and thus could not flag the application as suspicious. If eDoctor is deployed to a large number of users, it can learn an apps phases using many different instances of that app. eDoctor could then leverage this large data set to identify even "early onset" ABD issues.

eDoctor is meant to be used as a diagnosis tool instead of a detection tool. When the user observes a battery drain and invokes eDoctor, it reports the app that is most likely to be the root cause. So we focused the evaluation on correct diagnosis vs. misdiagnosis instead of true positives vs. false positives.

RTV vs. RUV. As expected, RUV is more accurate than RTV; the former had an accuracy of 94%, but the latter only diagnosed 72% of cases correctly. 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 overuse and other cases. RUV performs better than RTV in both categories. Interestingly, RTV is better at resource overuse (80.5%) than others (33.3%). The reason is that resource overuse often involve an app intensively using only one type of resource.

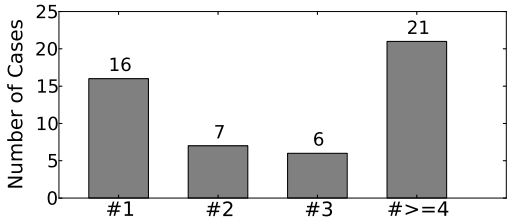
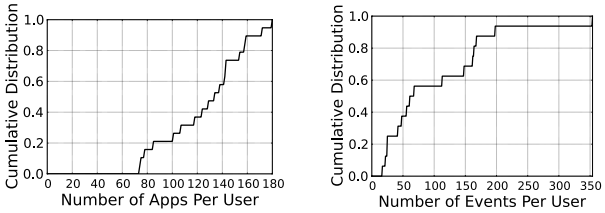


Figure 6: **Energy consumption rank of the culprit app**. The number at the top indicates the number of ABD cases, e.g., in 21 cases the rank is equal to or greater than 4.

Is the culprit app always the biggest energy consumer? As discussed in Section 1.2, one may wonder if existing energy profilers can detect ABD simply by identifying the top energy-consuming apps. Our data explains why this will not work. As illustrated in Figure 6, only 32% (16) of the cases have a culprit app that ranked #1 in energy use. In almost half (21) of the cases, the rank of the culprit app was greater than three. In these cases, the apps with ABD drained a significant amount of energy; however, other healthy, concurrently running apps also drew large amounts of energy (or the user noticed the ABD before the faulty app could waste a lot of energy). This demonstrates why existing profiling tools are insufficient for diagnosing many types of ABD. In addition, users may

be confused by the top-ranked but healthy apps and make wrong decisions to uninstall them or stop using them.

How many apps were monitored and how many events happened? As Figure 7 (a) shows, for at least 60% of the users, more than 120 apps are installed. The app counts include pre-installed apps and services that users were not even aware of. We also found that many energy-related events happened on the phone during the user study time period (7–10 days). As Figure 7 (b) shows, 60% of the users had at least 50 events taking place. As shown, eDoctor could diagnose culprits among all these events and monitored apps with high accuracy.



(a) (b)

Figure 7: **Distribution of the number of apps and events.**

5.1.2 Phase Distribution

To further understand the phase behavior of smartphone apps, we also examined how many normal phases smartphone apps may have. Figure 8 shows the cumulative distribution of all 1,890 apps that we monitored during the user study. The most important observation is that, during normal execution, most apps have a small number of major phases. 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 80% of the apps have at most 4 different phases. Section 3 described how eDoctor normalizes RUV with respect to CPU time. Figure 8 depicts the number of phases detected with and without normalization. As shown, normalization reduces the number of phases. After normalizing, nearly 75% of apps have only 1 normal phase.

5.2 Overhead

eDoctor's Information Collector periodically runs in the background (by default, once every 5 minutes). In this section, we describe eDoctor's overhead in terms of energy, storage, and memory.

Battery consumption overhead. We directly measured eDoctor's battery consumption on the Nexus One phone. 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

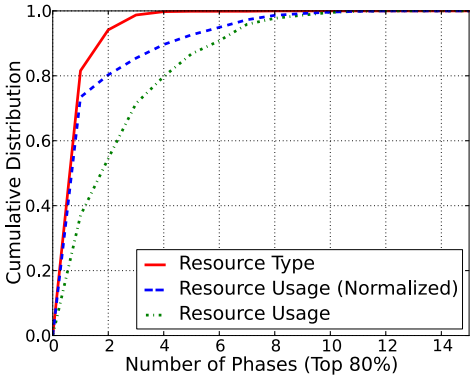


Figure 8: **The cumulative distribution of number of phases across 1,890 apps we monitored on real user phones during the user study**. We only consider the major phases that account for 80% of the total execution time.

in Figure 9, 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. eDoctor's energy overhead should be even lower—normal user activity will wake the phone up, allowing eDoctor to "piggyback" on this energy usage and collect resource statistics in the background. eDoctor's resource collection also has low overhead because eDoctor leverages Android's preexisting infrastructure for persistent resource tracking.

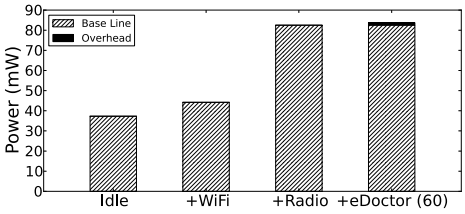


Figure 9: **eDoctor's battery consumption overhead for data collection**. Baseline (the first three bars): idle Nexus One phone with Wi-Fi and radio signal enabled. eDoctor collects all 60 active apps' resource usage on this phone (the fourth bar).

Storage overhead. eDoctor uses storage to collect resource utilization data and phase statistics. We measured this storage overhead by running a phone with eDoctor installed for 24 hours. eDoctor's overhead increases with the number of apps, so we ran experiments with 100, 125, and 150 installed apps. Table 4 shows that eDoctor consumed at most 3.2 MB per day. By default, eDoctor tracks one week of information; thus, eDoctor requires at most 22.4 MB of total storage. The storage overhead is even smaller in reality, because eDoctor does not store resource data if an app is not running. This is an acceptable overhead, since modern smartphones contain several gigabytes of storage space.

Number of Apps	100	125	150
Data size (24 hours)	1915 KB	2419 KB	2884 KB
Phase information	216 KB	270 KB	324 KB
Total	**2131 KB**	**2689 KB**	**3208 KB**

Table 4: **Storage used by eDoctor**

Memory Overhead. We used Trepn™ Profiler [3] to measure eDoctor's memory overhead. eDoctor's memory footprint was only 23.3 – 25.2MB. Memory utilization was stable over time because eDoctor only buffers a small amount of data and periodically stores the data to persistent storage.

6 Limitations and Discussions

When does eDoctor struggle? eDoctor has poor accuracy if the phases related to ABD were also common before the ABD began. This might happen if an application was initially started in a broken state (see Section 5.1.1). eDoctor's diagnostic accuracy will also suffer if a user simultaneously installs or reconfigures two apps, one of which is normal but energy-hungry, and another which has an ABD bug. In this scenario, eDoctor will regard both apps as suspect; since eDoctor only reports the highest ranked app, misdiagnosis may result. Such a situation did not arise during our user study, but finding automatic resolutions for such problems is an area for future work. Finally, misdiagnosis might also occur if the causal event occurred so long ago that it no longer resides in eDoctor's historical data.

Alternative approaches? While eDoctor leverages phase behavior to identify abnormal apps, but there are alternative approaches. For example, using signal processing techniques, one could detect abnormal energy consumption in the same way that network intrusion detectors identify traffic flows [9]. However, such techniques often have many false positives. One could also use dynamic bug detectors to identify code paths that may lead to ABD [22][33]. However, they introduce significant overhead because of the run-time instrumentation, which makes it hard to deploy directly to users' smartphones. In addition, they only work for ABD issues caused by already known bug patterns. In comparison, eDoctor is light weight and it can diagnose ABD issues caused by various types of misconfiguration, bugs and so on.

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—we could record resource usage without users having to jailbreak their phones. We could also modify the platform to track app-specific configuration changes.

7 Related Work

Energy consumption modeling and measurement. Carroll et al. [13] measured power consumption of components in modern smartphones. Thiagarajan et al. [48] measured energy used by mobile browsers. Shye et al. [46] studied how user activities affect battery consumption, and derived a linear regression based power model. Zhang et al. [52] presented a power model based on system variables, e.g., the processor's frequency, the amount of data received through the network, and the display brightness. Recently, Pathak et al. [35, 36] proposed "Eprof", a tool that performs fine-grained energy profiling by tracing system calls. eDoctor leverages Android's internal energy usage tracker; this tracker has less overhead, but it is sufficiently accurate for eDoctor to effectively rank applications by their energy usage.

Malware detection by monitoring energy usage. Kim et al. [25] detects malware that causes sudden battery drain as a side-effect. Similar to malware detection for desktops/laptops, this work detects *"known"* battery-drain malware by comparing the power signature of each application in a smartphone with those known signatures stored in a malware database. eDoctor focuses on diagnosing battery-drain caused by *unknown* software bugs or configuration changes that may happen to any smartphone apps.

Energy-efficient smartphone design. Prior work covers a wide spectrum of system design: processors (GreenDroid [21]), resource management (ECOSystem [51], Cinder [42]), file systems (quFiles [49]), page allocation ([29]), I/O interfaces (Co-op I/O [50]), display ([5]), wireless networking (PGTP [4], STPM [6], SleepWell [32], SALSA [41], Bartendr [43]), and high level services (EnLoc [14], Micro-blog [20], EnTracked [26], A-loc [31]). These efforts focus on reducing energy usage in *normal* circumstances. In comparison, eDoctor troubleshoots abnormal battery drain.

Abnormal battery drain studies. Pathak et al. [34] conducted a study on battery issues on Android. Their results also show that ABD is an emerging problem. Recently, Pathak et al. [37] studied energy bugs in smartphone apps. They found many bugs are caused by failing to release resources and thus preventing a phone from switching to sleep mode (called "NoSleep" bugs). They also proposed a detector leveraging a reaching definition data flow analysis to detect the missed API calls that release resources. Their work can help developers to detect this specific type of energy bug in source code. eDoctor is complementary because (1) eDoctor helps *users* diagnose ABD issues and find the appropriate repairs; and (2) eDoctor does not assume that ABD is caused by specific types of bugs. Instead, eDoctor can diagnose ABD that arises from a variety of misconfigurations, bugs and so on.

8 Conclusions

This paper addresses the emerging abnormal battery drain (ABD) issue on smartphones. We built a practical tool, eDoctor, to help users diagnose and repair ABD issues. 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 plan to release eDoctor on Google Play so that it can help real users while also collecting feedback for further improvement.

Acknowledgments

We greatly appreciate NSDI anonymous reviewers for their insightful feedback. We especially thank our shepherd, Dr. James W. Mickens, for his great effort to help us improve the paper. We also thank the members in the OPERA research group and the UCSD System and Networking (SysNet) group, and Erik Hinterbichler for their discussions and paper proof-reading. Last but not the least, we are grateful to the volunteers who participated our user study. This research is supported by NSF CNS–0720743 grant, NSF CSR Small 1017784 grant and NSF CSR–1217408 grant.

References

[1] Android 1.6 Platform Highlights - Battery Usage Indicator. http://developer.android.com/about/versions/android-1.6-highlights.html.

[2] App Store (iOS). http://en.wikipedia.org/wiki/App_Store_(iOS).

[3] Trepn ™Profiler. https://developer.qualcomm.com/mobile-development/development-devices/trepn-profiler.

[4] ANAND, B., SEBASTIAN, J., MING, S., ANANDA, A., CHAN, M., AND BALAN, R. Pgtp: Power Aware Game Transport Protocol for Multi-player Mobile Games. In *Proceedings of the International Conference on Communications and Signal Processing* (2011), ICCSP '11, pp. 399–404.

[5] ANAND, B., THIRUGNANAM, K., SEBASTIAN, J., KANNAN, P. G., ANANDA, A. L., CHAN, M. C., AND BALAN, R. K. Adaptive Display Power Management for Mobile Games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 57–70.

[6] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Self-tuning Wireless Network Power Management. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking* (2003), MobiCom '03, ACM, pp. 176–189.

[7] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Ghosts in the Machine: Interfaces for Better Power Management. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services* (2004), MobiSys '04, ACM, pp. 23–35.

[8] BALASUBRAMANIAN, N., BALASUBRAMANIAN, A., AND VENKATARAMANI, A. 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* (2009), IMC '09, ACM, pp. 280–293.

[9] BARFORD, P., KLINE, J., PLONKA, D., AND RON, A. A Signal Analysis of Network Traffic Anomalies. In *Internet Measurement Workshop* (2002), pp. 71–82.

[10] BICKFORD, J., LAGAR-CAVILLA, H. A., VARSHAVSKY, A., GANAPATHY, V., AND IFTODE, L. Security Versus Energy Tradeoffs in Host-based Mobile Malware Detection. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 225–238.

[11] CALIMLIM, A. Apple iOS 6.1 Reportedly Plagued With Battery, 3G and Syncing Issues. http://mashable.com/2013/02/09/ios-6-1-issues/, February 2013. by Mashable.

[12] CANALYS. Smartphones Overtake Client PCs in 2011. http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011, February 2012.

[13] CARROLL, A., AND HEISER, G. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the USENIX Annual Technical Conference* (2010), USENIX ATC'10, USENIX Association, pp. 21–21.

[14] CONSTANDACHE, I., GAONKAR, S., SAYLER, M., CHOUDHURY, R., AND COX, L. EnLoc: Energy-efficient Localization for Mobile Phones. In *Proceedings of the 28th Conference on Computer Communications* (2009), INFOCOM '09, IEEE Computer Society, pp. 2716–2720.

[15] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010), MobiSys '10, ACM, pp. 49–62.

[16] DHODAPKAR, A. S., AND SMITH, J. E. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), MICRO '36, IEEE Computer Society, pp. 217–228.

[17] DONG, M., AND ZHONG, L. Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 85–98.

[18] DONG, M., AND ZHONG, L. 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* (2011), MobiSys '11, ACM, pp. 335–348.

[19] DUESTERWALD, E., CASCAVAL, C., AND DWARKADAS, S. Characterizing and Predicting Program Behavior and Its Variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (2003), PACT '03, IEEE Computer Society, pp. 220–231.

[20] GAONKAR, S., LI, J., CHOUDHURY, R. R., COX, L., AND SCHMIDT, A. Micro-blog: Sharing and querying content through mobile phones and social participation. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (2008), MobiSys '08, ACM, pp. 174–186.

[21] GOULDING-HOTTA, N., SAMPSON, J., VENKATESH, G., GARCIA, S., AURICCHIO, J., HUANG, P.-C., ARORA, M., NATH, S., BHATT, V., BABB, J., SWANSON, S., AND TAYLOR, M. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro 31* (March 2011), 86–95.

[22] HANGAL, S., AND LAM, M. S. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering* (2002), ICSE '02, ACM, pp. 291–301.

[23] HUANG, M. C., RENAU, J., AND TORRELLAS, J. Positional Adaptation of Processors: Application to Energy Reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (2003), ISCA '03, ACM, pp. 157–168.

[24] JUNE, L. Facebook Mobile App Stats Shocker: 104M iPhone Users, 12M Android Users.

http://www.engadget.com/2010/08/25/facebook-mobile-app-stats-shocker-104-million-iphone-users-12, August 2010. from engadget.

[25] KIM, H., SMITH, J., AND SHIN, K. G. Detecting Energy-greedy Anomalies and Mobile Malware Variants. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (2008), MobiSys '08, ACM, pp. 239–252.

[26] KJÆGAARD, M. B., LANGDAL, J., GODSK, T., AND TOFTKJÆR, T. EnTracked: Energy-efficient Robust Position Tracking for Mobile Devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services* (2009), MobiSys '09, ACM, pp. 221–234.

[27] LATEROID. JuiceDefender, an Battery Saving Application on Android. `http://latedroid.com/juicedefender`.

[28] LAU, J., PERELMAN, E., HAMERLY, G., SHERWOOD, T., AND CALDER, B. 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, IEEE Computer Society, pp. 135–146.

[29] LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. Power Aware Page Allocation. *SIGARCH Comput. Archit. News 28*, 5 (Nov. 2000), 105–116.

[30] LIN, F. X., WANG, Z., LIKAMWA, R., AND ZHONG, L. Reflex: Using Low-power Processors in Smartphones Without Knowing Them. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS '12, ACM, pp. 13–24.

[31] LIN, K., KANSAL, A., LYMBEROPOULOS, D., AND ZHAO, F. Energy-accuracy Trade-off for Continuous Mobile Device Location. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010), MobiSys '10, ACM, pp. 285–298.

[32] MANWEILER, J., AND ROY CHOUDHURY, R. Avoiding the Rush Hours: WiFi Energy Management via Traffic Isolation. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 253–266.

[33] NETHERCOTE, N., AND SEWARD, J. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), PLDI '07, ACM, pp. 89–100.

[34] PATHAK, A., HU, Y. C., AND ZHANG, M. Bootstrapping Energy Debugging on Smartphones: a First Look at Energy Bugs in Mobile Devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), HotNets-X, ACM, pp. 5:1–5:6.

[35] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones With Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), pp. 29–42.

[36] PATHAK, A., HU, Y. C., ZHANG, M., BAHL, P., AND WANG, Y.-M. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proceedings of the 6th ACM European Conference on Computer Systems* (2011), EuroSys '11, ACM, pp. 153–168.

[37] PATHAK, A., JINDAL, A., HU, Y. C., AND MIDKIFF, S. P. 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* (2012), MobiSys '12, ACM, pp. 267–280.

[38] PERELMAN, E., HAMERLY, G., AND CALDER, B. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (2003), PACT '03, IEEE Computer Society, pp. 244–255.

[39] PERRUCCI, G., FITZEK, F., SASSO, G., KELLERER, W., AND WIDMER, J. On the Impact of 2G and 3G Network Usage for Mobile Phones' Battery Life. *Wireless Conference* (2009), 255–259.

[40] POWERS, R. Batteries for Low Power Electronics. *Proc. of the IEEE 83*, 4 (April 1995), 687–693.

[41] RA, M.-R., PAEK, J., SHARMA, A. B., GOVINDAN, R., KRIEGER, M. H., AND NEELY, M. J. Energy-delay Tradeoffs in Smartphone Applications. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010), MobiSys '10, ACM, pp. 255–270.

[42] ROY, A., RUMBLE, S. M., STUTSMAN, R., LEVIS, P., MAZIÈRES, D., AND ZELDOVICH, N. Energy Management in Mobile Devices with the Cinder Operating System. In *Proceedings of the 6th ACM European Conference on Computer Systems* (2011), EuroSys '11, ACM, pp. 139–152.

[43] SCHULMAN, A., NAVDA, V., RAMJEE, R., SPRING, N., DESHPANDE, P., GRUNEWALD, C., JAIN, K., AND PADMANABHAN, V. N. Bartendr: a Practical Approach to Energy-aware Cellular Data Scheduling. In *Proceedings of the 16th Annual International Conference on Mobile Computing and Networking* (2010), MobiCom '10, ACM, pp. 85–96.

[44] SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), ASPLOS '02, ACM, pp. 45–57.

[45] SHERWOOD, T., SAIR, S., AND CALDER, B. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (2003), ISCA '03, ACM, pp. 336–349.

[46] SHYE, A., SCHOLBROCK, B., AND MEMIK, G. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), MICRO 42, ACM, pp. 168–178.

[47] SMITH, C. Google Play: 20 Billion App Downloads, 600,000 Apps and Games. http://www.androidauthority.com/google-play-20-billion-app-downloads-600000-apps-games-98006/, June 2012. by AndroidAuthority.

[48] THIAGARAJAN, N., AGGARWAL, G., NICOARA, A., BONEH, D., AND SINGH, J. P. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proceedings of the 21st International Conference on World Wide Web* (2012), WWW '12, ACM, pp. 41–50.

[49] VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. quFiles: the Right File at the Right Time. *Trans. Storage 6* (September 2010), 12:1–12:28.

[50] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: a Novel I/O Semantics for Energy-aware Applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002), OSDI '02, ACM, pp. 117–129.

[51] ZENG, H., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. ECOSystem: Managing Energy as a First Class Operating System Resource. *SIGOPS Oper. Syst. Rev. 36* (October 2002), 123–132.

[52] ZHANG, L., TIWANA, B., QIAN, Z., WANG, Z., DICK, R. P., MAO, Z. M., AND YANG, L. 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* (2010), CODES/ISSS '10, ACM, pp. 105–114.