eDelta: Pinpointing Energy Deviations in Smartphone Apps via Comparative Trace Analysis

Li Li[1], Bruce Beitman[1], Mai Zheng[2], Xiaorui Wang[1], and Feng Qin[1]

[1]The Ohio State University, Columbus, OH, USA
[2]New Mexico State University, Las Cruces, NM, USA

***Abstract*—Many smartphone apps can consume an unnecessarily high amount of energy, shortening battery life. Although users can easily notice the undesired fast battery drain, it is almost impossible for them to precisely remember how the abnormal battery drain (ABD) is triggered, making it difficult for developers to fix the problem. Therefore, app developers are in an urgent need for a tool that can provide them helpful information.**

In this paper, we propose eDelta, a framework that assists developers in pinpointing the APIs with high energy deviation, which usually have a high probability of being relevant to the non-deterministic ABD. Specifically, eDelta performs comparative trace analysis to identify APIs that have significant energy consumption deviation in different user traces. With the information provided by eDelta, developers can substantially reduce the time they spend searching for the ABD root causes. We have prototyped eDelta in Android 4.4 and evaluated it with twenty real-world apps. Our results show that eDelta can effectively pinpoint the APIs with high energy deviation and those APIs indeed cause ABD. Specifically, it reduces, on average, 94.6% of the amount of code that the developers would need to search for ABD root causes.

I. Introduction

With the rapid growth of different apps on smartphone, software defects are becoming common in these apps. One main reason is that the learning curve of developing smartphone apps has been made low enough for many novice developers to freely release creative, but poorly written, software [1]. In a recent study [2], 40% of app developers are reported working completely on their own and 26% of app developers have less than 2 years of experience in writing apps.

Among the app defects, a new type called abnormal battery drain (ABD) has been troubling many smartphone users [3]. ABD refers to abnormally fast draining of a smartphone's battery which can be caused by different issues such as API [1] misuses (e.g., forget to call wakelock.release() API in certain code path, thus unnecessarily keep CPU awake) and inefficient design (e.g., an app keeps retrieving GPS information to render an invisible map when it is in the background) resulting in overusing the system resources. ABD issues have adversely affected app developers via negative user feedback. A recent user study [4] report 55% of users would give a bad review after experiencing heavy battery drain. Therefore, app developers are in an urgent need for a tool that can help them identify the ABD root causes.

[1]In this work, API refers to the programming interface provided by Android SDK framework, not user-defined functions.

While users can easily notice the ABD symptom, it is almost impossible for them to precisely remember how these energy anomalies are triggered, which makes it hard for developers to correct such ABD defects. For example, reports from users usually only describe symptoms and may not be helpful for the developers to even verify the existence of the issue [5]. Unfortunately, while there have been tools to detect ABD for phone users [3], to our best knowledge, there are few studies which can help developers fix ABD issues. In this work, we observe that an ABD is often triggered in a certain code path and by a particular user input, similar to other software defects [6]. Due to the non-deterministic nature of such software defects, the traces collected from different users may or may not have the ABD manifested. Thus, energy deviation exists for the same API across different traces and such deviation can be used for ABD detection.

Prior research on ABD detection can be categorized into two groups: 1) Source-based methods [6] (often performed in the testing phase before app release) that help developers analyze the app source code, and 2) trace-based methods [7] (often performed in the wild) that help app users detect which app causes ABD by diagnosing the usage traces. Most source-based methods are designed from the developers' perspective. For example, no-sleep ABD can be detected by analyzing when the wakelock is acquired and released [6]. While source-based methods are powerful and can even detect ABD cases that rarely manifest, they are usually designed specially for a particular type of ABD (e.g., no-sleep). Therefore, ABD with previously unknown causes is hard to diagnose via this method. Existing work on trace-based detection focuses mainly on helping users (instead of developers) detect which app causes ABD, and are therefore often coarse-grained [3], [7]. For example, eDoctor [3] is designed to "identify the *problematic app* for an ABD issue". eDoctor detects the app that causes ABD and provides the corresponding repair solution (e.g., uninstall the app) to users. Although such tools can indeed help users, the information they provide is too coarse-grained and therefore is insufficient for developers to fix the ABD problem in their code.

In this paper, we propose eDelta, an automated trace-based detection framework that assists app developers in pinpointing the APIs with high energy deviation through comparatively analyzing the collected traces. Certain APIs maintain high power consumption in some traces and low power consump-

978-1-5386-3470-7/17/$31.00

tion in other traces. These APIs exhibit energy deviation and usually have a high probability of being relevant to the ABD. This observation is intuitive [8] and has been verified by our evaluation results. Thus, with the reported information, developers can directly go to the pinpointed code segments to fix the ABD problem.

In particular, eDelta consists of two parts: 1) online tracking on smartphones, and 2) offline diagnosis on a remote server. For online tracking, eDelta first instruments selected APIs. Then, the instrumented app is provided to users for collecting usage traces. Since traces can come from phones with different hardware and software configurations, we adopt the model scaling technique proposed in [9] to make their power data comparable. After trace collection, eDelta starts offline diagnosis that features *a trace segmentation approach and a statistical analysis algorithm*. For effective analysis, we separate the collected traces into different segments based on their usage scenarios (e.g., foreground/background). After that, eDelta checks if the power consumption of an instrumented API from some traces is much higher than, i.e., deviates from, that of the same API in other traces under the same usage scenarios. If yes, this API has a high probability of being relevant to the ABD problem. If such APIs cannot be successfully identified with sufficient confidence, the ABD might be caused by some APIs that are not instrumented. Developers can then instrument more APIs and loop back to the online tracking part for collecting more traces. This iterative process finally identifies more APIs with high energy deviation that helps developer to fix the ABD.

In summary, this paper makes the following major contributions:

- We propose eDelta, a framework that assists app developers in pinpointing the APIs with high energy deviation.
- We design a trace segmentation approach that separates the collected traces into different usage scenarios for effective analysis of different APIs.
- We design a statistical algorithm to detect the APIs with high energy deviation.
- We have prototyped eDelta in Android. Our results show that the APIs with high energy deviation reported by eDelta can usually effectively help developers fix the ABD caused by different issues. Overall, eDelta can reduce 94.6% of the amount of code that the developer would need to search for fixing the ABD.

The rest of this paper is organized as follows. Section II discusses the related research. Section III introduces the design of eDelta. Section IV presents our evaluation results. Section V concludes the paper.

II. Related Work

Our work is closely related to two major research topics, *energy bug detection* and *power modeling and monitoring*.

Method	Detect ABD Apps	Detect API Misuses
Trace-based Solution	[3], [7], [10]	N/A
Source-based Solution	N/A	[6], [11]

TABLE I: Existing work related to energy bug detection.

Energy Bug Detection. As shown in Table I, the existing work can be mainly divided into two categories: 1) Detecting which app causes the ABD, and 2) detecting a particular type of API misuse. For the first category, eDoctor [3] tries to cluster the app execution of a user's phone into different phases based on the resource utilization. While the phase detection is effective in identifying the abnormal app, it is too coarse grained for developers to diagnose the root cause of ABD within an app. Oliner et al. [7] design a collaborative tool to detect energy bugs and energy hogs. All these approaches in the first category aim to help phone users by detecting the ABD app. In contrast, eDelta helps developers to fix the ABD problem in their app code through pinpointing the APIs with high energy deviation.

For the second category, different approaches have been proposed to detect the wakelock related energy bug through analyzing source code and pinpointing wakelock related API misuses [6], [11]. Since these solutions are designed for detecting a particular type of ABD (e.g., no sleep), they can hardly detect the ABDs with previously unknown causes. Unlike these approaches, the APIs with high energy deviation reported by eDelta can help developers fix the ABD caused by different issues such as API misuse and inefficient design.

Power Modeling and Monitoring. Recent studies have proposed different power modeling methodologies. Yoon et al. [12] have presented AppScope, which monitors an app's hardware usage at the kernel level. Zhang et al. [13] have designed PowerTutor, an online power estimation tool. Pathak et al. [8] have proposed Eprof, an energy profiler based on a finite state machine power model. Although energy characterization of mobile devices has improved greatly, they focus on a different purpose and do not directly provide information about ABD to developers. This is because high energy consumption is not necessarily energy misuse. There is not a priori specification of abnormal battery drain. In contrast, we detect power deviations through comparatively analysis which have higher probability of being relevant to ABD.

III. Design and Implementation

A. Overview

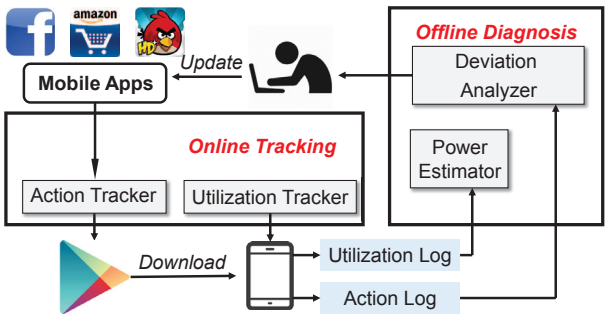

Fig. 1: System Architecture of eDelta.

1) System Architecture: Figure 1 shows the architecture of eDelta. It mainly consists of two parts: 1) online tracking on smartphones and 2) offline diagnosis on a remote server. The

online tracking part includes two main components, *Action Tracker* and *Utilization Tracker*. Action Tracker records the specific API calls from the suspect app in real time. Utilization Tracker records the utilization of system components during the execution of the suspect app. Offline diagnosis contains two main components, including *Power Estimator* and *Deviation Analyzer*. Power Estimator calculates the power consumed by each instrumented API in the suspect app at the thread level. Deviation Analyzer first separates the traces into different usage scenarios and then adopts a statistical approach to pinpoint the APIs with high energy deviation. This is based on the observation that the collected traces often contain different usage scenarios (e.g., interact with the app in the foreground, put the app in the background). In Android, an action (API) usually maintains different behaviors within different usage scenarios. For instance, in some navigation apps, LocationManager (API provided by Android system to access the location service) consumes about 500 mW when the app is in the foreground (GPS is turned on to render user's current location on the map), while it consumes 0 mW when the app is switched to the background (GPS is turned off). We assume that an API should consume similar power within the same scenario across different traces. The APIs that maintain high energy deviation within the same scenario have a high probability of being relevant to the ABD.

To better illustrate the framework, we briefly describe the workflow of eDelta as follows. When facing an ABD report of an app, the developer does not know where to fix the ABD issue in the app code. So the developer first instruments the app with eDelta's Action Tracker. To reduce the runtime overhead caused by logging, eDelta *selectively* instruments related APIs that are not excessively fine grained, yet can sufficiently reveal important diagnosis information. After that, when users run the instrumented apps on their phones, the invocations of selected APIs and utilization of system hardware components are logged into two trace files, respectively. The instrumented version does not impact user perceived usage experience and the performance overhead is negligible according to our evaluation results. Next, the collected traces (usage traces and system utilization traces) are sent to a remote server when the smartphone is being charged and the WiFi is active. Thus the communication overhead can be negligible. After receiving the logs, a diagnosis engine running on the back-end server identifies and reports APIs (and corresponding class name in which the APIs are called) that exhibit high energy deviation. Developers can then directly go to the relevant code segments to fix the ABD problem.

Specifically, eDelta provides the following information to help developers correct the ABD. First, it reports the APIs with high energy deviation. Second, it reports the power and corresponding action traces.

2) A Motivation Example: As mentioned above, abnormal battery drain can be caused by different issues (e.g., API misuses in certain code paths, or app design problem) that manifest energy deviation across different traces. *We believe the APIs with significant energy deviation usually contain useful information for developers to fix the ABD problem.* eDelta comparatively analyzes traces under different contexts from different users (usually contains traces with and without the ABD being manifested) to help developers pinpoint the APIs with high energy deviation. We briefly introduce the comparative analysis eDelta performs as follows.

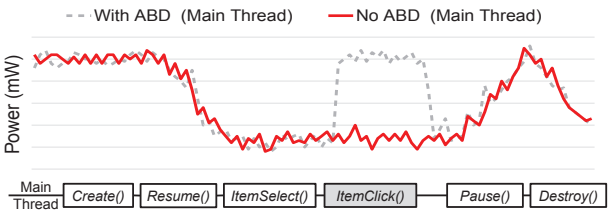

Fig. 2: A simplified example of the comparative analysis from two user traces, with and without ABD manifestation. The shaded API (*ItemClick()*) causes the unusual high energy consumption.

Figure 2 shows a simplified example of the analysis. The traces represent power consumption of those instrumented APIs in the main thread of the reported app. The trace represented by dotted line manifests an ABD, which can be caused by various factors such as misconfiguration. The trace represented by the solid red line does not manifest the ABD. We can see that the instrumented API *ItemClick()* is attributed with additional power in the trace with ABD manifested. eDelta reports *ItemClick()* to maintain deviation through comparatively analyzing the thread-level power consumption from different traces.

Cross-trace analysis is adopted for two reasons. First, it relaxes the requirement that the collected traces must contain the transition point from normal to abnormal, which is required if a single trace is used for analysis. For eDelta, as long as the traces impacted by the ABD and the traces without ABD manifested are collected, meaningful results can be generated through comparing the power consumption of action instances in different traces. In contrast, existing solutions (e.g., [3]) must rely on traces that contain the manifestation point. Second, cross-trace analysis can make use of larger amounts of data, because even those traces with no ABD manifested can be used for comparison purpose. The more data is used in the analysis, the more reliable information can be reported to developers.

B. Online Tracking

eDelta dynamically collects the information across different users under different contexts. *Action Tracker* and *Utilization Tracker* are designed to record the runtime information from different perspectives.

Action Tracker. Action Tracker records specific API calls from the suspect app. To minimize the runtime logging overhead, eDelta selectively instruments two types of API calls, user-app interaction APIs and hardware related APIs. In general, for modern smartphones the major power consuming components (e.g., GPS [8]) are usually accessed via of a set of APIs provided by the Android SDK framework (hardware

related APIs) in the app code. Moreover, these components are often triggered by certain user inputs (handled by interaction related APIs). Guided by this rationale, we create a pool of the selected APIs that need to be instrumented.

Table II shows the examples of commonly used APIs in the pool. For each invocation of the instrumented API, action tracker records the system timestamps at the start and end of the API invocation, along with the API name. Specifically, for callback functions related to user interactions, such as onClick(), eDelta takes two time stamps at the entrance and exit points of the callback function. For hardware-related components, there often exists a pair of APIs to invoke the component and to stop the component. For instance, Wakelock.acquire() and Wakelock.release() are used to activate and deactivate a wakelock respectively. Thus, system timestamps when these two APIs are invoked are treated as the starting point and ending point of the corresponding action (e.g., Wakelock). This information is logged in the app action log.

Figure 3 shows an example action log when a user uses the Facebook app. The number at the beginning of each line shows the timestamp. "+" represents the entrance point of the function and "-" represents the exit point of the function. Moreover, name of the class in which the action is invoked is also recorded, such as "Lcom/facebook/katana/binding/AppSession".

Category	API Class Name	Example APIs
Activity Life Cycle Related (I)	android.app.Activity	onCreate, onStart, onResume, onPause, onStop, etc.
UI Related (I)	android.View	onClick, onLongClick, onKey, onTouch, etc.
CPU (H)	android.os.AsyncTask android.os.PowerManager	doInBackground, get, cancel, etc. partial_wakelock, screen_dim_wake_lock, etc.
Screen (H)	android.os.PowerManager	partial_wakelock, full_wake_lock, etc.
Sensors (H)	android.hardware.SensorManager	getOrientation, registerListener etc.
Camera (H)	android.hardware.Camera	open, release, takePicture, etc.
Data Storage (H)	android.preference.PrefManager	getSharedPreferences etc.
WiFi (H)	android.net.wifi.WifiManager	WifiLock.acquire, createWifiLock, etc.

TABLE II: Categories and examples of APIs that are instrumented to gather action usage (*H* represents hardware related APIs and *I* represents interaction related APIs).

```
108757 + Lcom/facebook/katana/SyncContactsSetupActivity; onClick
108757 - Lcom/facebook/katana/SyncContactsSetupActivity; onClick
109107 + Lcom/facebook/katana/service/FacebookService; onStart
109108 + Lcom/facebook/katana/binding/AppSession; acquireWakeLock
109112 - Lcom/facebook/katana/service/FacebookService; onStart
111190 - Lcom/facebook/katana/binding/AppSession; releaseWakeLock
113357 + Lcom/facebook/katana/FacebookActivity; onPause
113358 - Lcom/facebook/katana/FacebookActivity; onPause
```

Fig. 3: An example action log when a user uses Facebook app. User interaction related actions (onPause(), onClick()) and hardware related actions (Wakelock) are recorded.

Utilization Tracker. In order to estimate power consumption of the app, utilization tracker periodically records the utilization of system components (i.e. CPU, display, WiFi, etc.). It monitors proc filesystem (*procfs*) to gather hardware utilization assigned to the target app and provides the utilization for each thread of the app, which allows eDelta to estimate the power consumption of each thread individually. Thread level utilization tracking limits to CPU activities. Moreover, *procfs* reports how many CPU cycles are idling, which allows eDelta to account for *wakelock* actions (a mechanism of power management service in Android OS, which can be used to keep CPU awake and keep the screen on).

Utilization tracker is implemented as a background service on Android. The utilization tracking is limited only to the suspect app identified by its PID, and its corresponding threads identified by their TIDs. Thus, existence of multiple running apps does not affect utilization tracking of the suspect app. Timestamp of each sampling point is recorded to estimate the power consumption of each instrumented API in Power Estimator in Section III-C.

C. Offline Diagnosis

After collecting the runtime information, eDelta comparatively analyzes the traces to pinpoint the APIs with high energy deviation. Offline diagnosis mainly contains two components including *Power Estimator* and *Deviation Analyzer*.

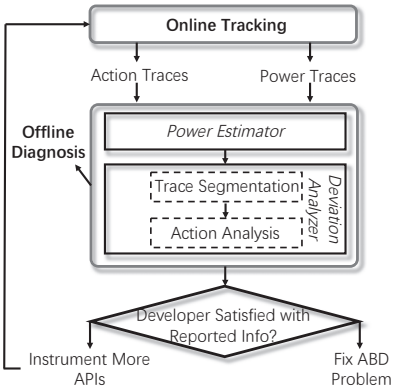

Fig. 4: Workflow of identifying energy deviation and pinpointing ABD.

1) Power Estimator: Power Estimator feeds the utilization values to a power model [8], [9], [12]–[16], corresponding to the type of the smartphone, to calculate the power consumed by each hardware component. It is important to note that Power Estimator uses the hardware utilization to generate power profile that contains power consumption of each instrumented API at the thread level. Specifically, user interaction related actions are attributed with the thread-level average power between the starting and ending points of the action. For hardware related actions, power consumption is attributed according to their utility. In other words, power consumption of a certain hardware component is attributed to the corresponding APIs that access it. For instance, average power of GPS between the time points when LocationManager.requestLocationUpdate() and LocationManager.removeLocationUpdate() are invoked is attributed to the LocationManager action.

2) Deviation Analyzer: Deviation analyzer reports the APIs with high energy deviation to developers through comparative trace analysis. It contains two main steps including *trace segmentation* and *action analysis*.

Action Examples	Segmentation Vector
onCreate [I]	<Display>
onClick [I]	<Display>
LocationManager [H]	<Display>
MediaPlayer [H]	<Display>
PARTIAL_WAKE_LOCK [H]	<GPS, Sensors, Audio, Network, Display>
SCREEN_DIM_WAKE_LOCK [H]	<GPS, Sensors, Audio, Network, Display>
FULL_WAKE_LOCK [H]	<GPS, Sensors, Audio, Network, Display>

TABLE III: Example of APIs and segmentation vector (H represents hardware related APIs and I represents interaction related APIs).

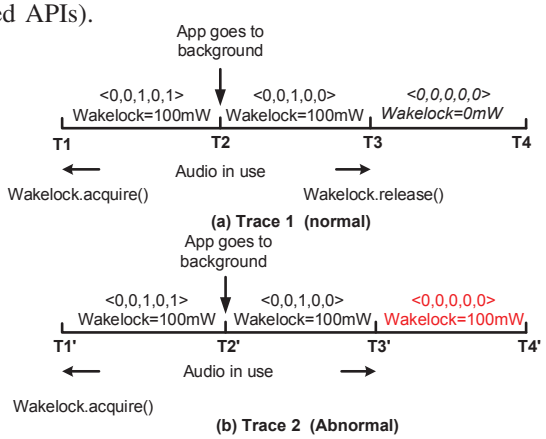

Fig. 5: Example traces for wakelock action analysis. Vector <GPS, Sensors, Audio, Network, Display> is adopted to separate traces into different execution segments. T1-T4 represent different time stamps.

Trace Segmentation. As mentioned earlier, the collected traces often contain different usage scenarios and need to be separated into different segments for effective intra-segment analysis. We observe that different usage scenarios often access different types of system resources. For instance, when a user reads an article with Facebook app, only the display is used in most of the time. On the other hand, network is usually utilized to retrieve data from remote servers when loading new web pages. System resource usage is then adopted as a metric for trace segmentation. The trace segments that maintain the same resource usage type are considered as the same segment. A usage vector is utilized to represent the resource usage situation. In the vector, each bit shows the usage of a corresponding component and is represented as a binary variable. When a component is utilized, the corresponding bit is set to 1, otherwise it is set to 0. For instance, <Display> shows an example vector which represents the display usage information of an app. Thus, a segment when an app is using the display (the app is in the foreground) can be represented as <1>. When the app is switched to the background (the app is not using display), the corresponding segment can be represented as <0>.

As discussed in Section III-B, eDelta instruments two types of APIs including hardware-related APIs and user interaction related APIs. For the analysis of these APIs (e.g., LocationManager, Media Player, onClick()), vector <Display> is used to differentiate app execution segments. This is because usage behavior of these components are often impacted by whether the app is in foreground (uses display). For most apps, users

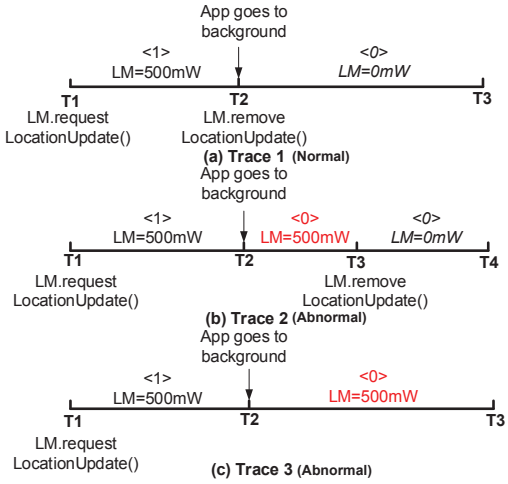

Fig. 6: Example traces for LocationManager action analysis. Vector <Display> is adopted to separate traces into different execution segments. T1-T4 represent different time stamps.

interact with and retrieve information from them through the display when they are in foreground (e.g., GPS is turned on to render location information). The app is usually in the sleep state when switched to background (e.g., GPS is turned off). Thus, display is an important factor that impacts the app behaviors. The action instances of a certain action under the two segments ((1) <1> the app is in the foreground and uses the display, (2) <0> the app is in the background and does not use the display) are retrieved.

It is important to note that for the analysis of WakeLock related APIs (e.g., PARTIAL_WAKE_Lock, etc), we treat them in a different way. Vector <GPS, Sensors, Audio, Network, Display> is adopted for trace segmentation. This is because Wakelock is often used to prevent devices from falling asleep during critical computation. Take networking and communications for example, many apps frequently fetch data from remote servers and present them to users. These tasks typically should not be disrupted by device sleeping when users are using the apps and expecting to see certain updates. Wakelocks are needed in such scenarios. Thus, wakelock is often correlated with other hardware components [17]. GPS, sensors, audio, network and display are selected is based on the observation that these five components are major components on smartphones and are widely utilized in different apps [3]. The vector can be easily extended by adding other components if needed. Table III shows the summary of example actions (APIs) and the corresponding trace segmentation vectors.

For hardware related actions such as Wakelock and LocationManager, there often exists a pair of APIs to activate and deactivate the corresponding components (e.g., LocationManager.requestLocationUpdate() and LocationManager.removeLocationUpdate()). Whenever the deactivating API is detected in the collected trace, the segmentation vector after that point is recorded and a marker action is added to show that the corresponding action consumes 0 mW under that segment. This is based on the rationale that developers often use the APIs according to the usage scenario of the

app. For instance, in some navigation apps, LocationManager consumes about 500 mW when the app is in the foreground, while it consumes 0 mW when the app is switched to the background (LocationManager.removeLocationUpdate() is invoked and GPS is turned off). In this case, a maker action will be added to show that the LocationManager consumes 0 mW when the app is in the background.

Figure 5 shows a simple example of trace segmentation in which two types of execution traces are collected. We can see from trace 1 that a user uses audio at timestamp T1. Wakelock is acquired at timestamp T1 to guarantee the normal usage. At timestamp T2, the app is switched to background. After a certain time period, the audio stops and the wakelock is released at timestamp T3. Trace 2 shows the same usage scenario without the Wakelock being released (e.g., due to an exception). In this example, we focus on the analysis of the Wakelock action. Thus, vector <GPS, Sensors, Audio, Network, Display> is adopted for segmentation. As shown in Figure 5, trace 1 is separated into three different execution segments including <0,0,1,0,1>, <0,0,1,0,0> and <0,0,0,0,0>. In this case, <0,0,1,0,1> represents the segment when an app uses audio and display (in the foreground). Action instances under different execution segments are treated as different actions. For instance, in trace 1, the Wakelock action (between Wakelock.acquire() invoked at timestamp T1 and Wakelock.release() invoked at timestamp T2) is divided into two actions under two execution segments. The corresponding power is then attributed. They are represented as WL=100mW<0,0,1,0,1> and WL=100mW<0,0,1,0,0> as shown in Figure 5 (a). According to the above discussion, after the WakeLock.release() API is invoked, a maker action WL=0mW is added and the execution segment <0,0,0,0,0> is recorded as the third action in trace 1. This shows that the Wakelock action consumes 0mW under the execution segment <0,0,0,0,0> in which the components in the vector are not utilized. The three actions retrieved from trace 2 include WL=100mW<0,0,1,0,1>, WL=100mW<0,0,1,0,0>, WL=100mW<0,0,0,0,0>.

Figure 6 shows another example of trace segmentation which is performed for the analysis of LocationManager action. According to Table III, vector <Display> is adopted to separate a trace into different segments. We can see from trace 1 that GPS is invoked at time stamp T1. The app is switched to background at time stamp T2 and GPS is released. After LocationManager.removeLocationUpdate() is invoked, a maker action is added as LM=0mW<0> to show LocationManager action consumes 0 mW when the app is not using the display under certain traces. Thus, the action instances retrieved from Trace 1 are LM=500mW<1> and LM=0mW<0>. Figures 6 (b) and (c) represent the scenarios that GPS is not correctly released and show the corresponding action instances. After trace segmentation, action instances under the same execution segments are obtained and sent as input to Action Analysis. It is important to note that the APIs with different parameters are treated as different actions.

Action Analysis. Action analysis is performed to help developers pinpoint the APIs with high energy deviation. Usually the APIs that exhibit deviation consume high power in some traces (e.g., ABD is manifested) and relatively low power in other traces (e.g., ABD is not manifested). As shown in Figure 7, raw data shows the power consumption of different actions in the Facebook application from 20 different user traces. We can see that the power consumption of action instances of Wakelock<0,0,0,0,0> from different traces can be clearly separated into two groups, while the power consumption of the other actions cannot be clearly separated. Thus, clustering analysis is adopted to group the power data of each action (API) within the same execution segments into two clusters, i.e., normal (lower value) and abnormal (higher value) ranges. K-medoids Partitioning Around Medoids (PAM) [18] is implemented to perform the clustering analysis, which has more tolerance to noise and outliers.

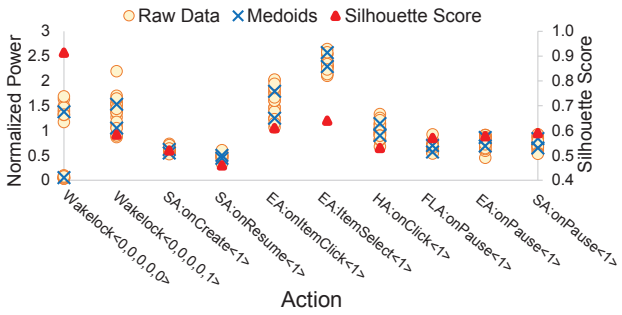

Fig. 7: Deviation analysis of Facebook app. Raw data represents the power consumption values of ten actions from 20 user traces. Medoids represents the center of each cluster. Silhouette score represents the validation of clustering analysis. The more confident the data is separated, the higher the silhouette score is.

In addition, Silhouette analysis [19] is performed to show how confident the data set can be separated into two clusters. The silhouette coefficient ranges from -1 to 1, where a larger value indicates that the object is well matched to its own cluster (cohesion) and badly matched to neighboring cluster (separation). Average silhouette score over all data of the entire data set is usually a measure of how appropriately the data have been clustered. According to [19], silhouette coefficient ranges from 0.71-1.0 indicates that a strong structure has been found in a data set. So we use 0.71 as the threshold. APIs with silhouette coefficients equal to or higher than this threshold will be reported to developers. Developers then can directly go to the pointed code segments to fix the ABD problem. Figure 7 shows the silhouette coefficient (right side of Y-axis) of different actions (APIs). We can see that Wakelock<0,0,0,0,0> (when the app is in the background and does not use other components) with a silhouette score higher than the threshold will be reported to developers.

The action and power traces are also reported to help developers further verify the ABD. Figure 8(a) shows the power traces with and without the ABD manifested. The trace with the ABD manifested represents approximately 100mW of extra power usage while the user is not using the app. Figure 8(b) shows the power breakdown with the ABD manifested.

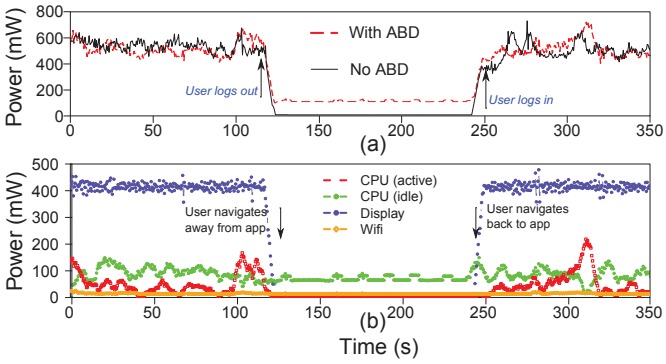

Fig. 8: Chronological traces of Facebook (a) total app power of traces (with same user inputs) with and without ABD manifested. (b) component power breakdown of the trace with the ABD manifested.

The *wakelock* is acquired at the beginning of the trace (around second 2), and keeps using the idle CPU energy while the user is not using the app. This leads to extra power consumption and causes the ABD. With the reported API with high energy deviation, the searching scope is reduced from 39,891 lines to 597 lines. After further investigating the pointed code segment, we confirm that the *wakelock* is kept indefinitely if the user does not log out. We can see that eDelta can efficiently reduce developer's searching scope of ABD root cause in their app code through reporting the APIs with high energy deviation.

IV. Evaluation

A. Experimental Methodology

We implement eDelta in Android 4.4. In our experiments, power model scaling [9] is adopted to make the power data collected from different smartphone models comparable. The deviation analysis is implemented in Python and R (a software environment for statistical computing [20]). Real-world phone usage and power traces are collected from different users under various contexts. We first discuss the overall results of applying eDelta to twenty different apps. Due to the limited space, we only present one case study in detail, followed by the performance and power overhead. For more detailed analysis of other cases, please refer to our technical report [21].

B. Overall Results

In this section, we discuss the overall results of applying eDelta to twenty different apps. Table IV shows the results when eDelta is applied to twenty ABD apps caused by different issues [1] including no-sleep issue, loop issue, configuration issue and immortality issue. No sleep issue erroneously does not allow at least one component of the phone to sleep, resulting in unnecessary battery drain. Loop issue happens where a part of an app performs periodic but unnecessary tasks. Configuration issue represents the misconfiguration of the application which results in high battery drain. Immortality issue is a situation where a buggy application that drains battery, upon being explicitly killed by the users, respawns and continues to drain battery.

App	Downloads	Root Cause	Code Reduction
Boston Bus Map	100k+	loop	89.4%
OwnCloud	500k+	loop	99.7%
Aagtl	50k+	configuration	96.2%
Sensorium	5k+	configuration	95.5%
Signal	140k+	configuration	99.1%
CommonsWare	N/A	immortality	87.1%
Facebook	1B+	no-sleep	98.5%
Open Camera	1M+	no-sleep	99.4%
droid VNC	500k+	no-sleep	96%
A Better Camera	1M+	no-sleep	95%
Binaural-Beats	50k+	no-sleep	97.4%
Ushahidi	50k+	no-sleep	93.7%
Sofia Navigation	5k+	no-sleep	97.9%
Osmdroid	50k+	no-sleep	89.2%
Geohashdroid	50k+	no-sleep	97.2%
Babblesink	N/A	no-sleep	86.2%
Traccar	50k+	no-sleep	92.5%
Tinfoil	500k+	no-sleep	93.6%
Pedometer	5k+	no-sleep	92.3%
FBReader	100k+	no-sleep	92.1%

TABLE IV: Apps used to evaluate eDelta. Code Reduction means the percentage of code lines that can be reduced for pinpointing the root cause of ABD.

We use code reduction as the metric to evaluate the effectiveness of eDelta. Each app's code reduction is calculated as $\frac{N_{Entire}-N_{Report}}{N_{Entire}}$. N_{Report} represents the number of byte-code lines responsible for the APIs that eDelta reports through the action analysis. N_{Entire} represents the entire code lines. We use byte-code lines because the source code of some apps is not publicly available. The last column of Table IV shows the code reduction of the corresponding apps. It shows that eDelta can reduce the code needed for diagnosis for up to 99.4% (94.6% on average), which implies that eDelta can significantly reduce developer's effort to fix the ABD.

C. Case Study: Boston Bus Map

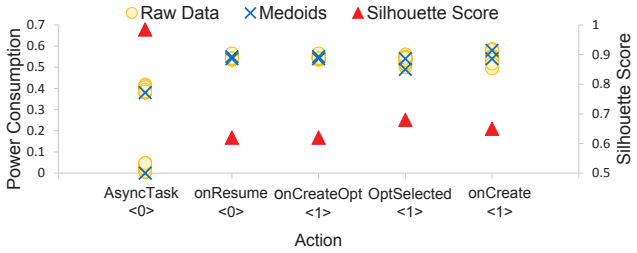

Fig. 9: Deviation analysis of Boston Bus Map. Raw data represents the power consumption of five actions from 38 different user traces. Medoids represents the center of each cluster. Silhouette Score represents the validation of clustering analysis.

Figure 9 shows the analysis result of the Boston Bus Map (an open-source app for locating buses in Boston) from 38 different user traces. We can see that power consumption of action AsyncTask<0> (when the app does not use the display) can be clearly separated into two groups. The silhouette coefficient is higher than the threshold. Thus, AsyncTask<0> is reported to the developer to exhibit high energy deviation. Figure 10(a) shows user traces with and without the ABD manifested of Boston Bus Map, which shows a spike in power between 24s and 30s (the user is not using the app). Figure

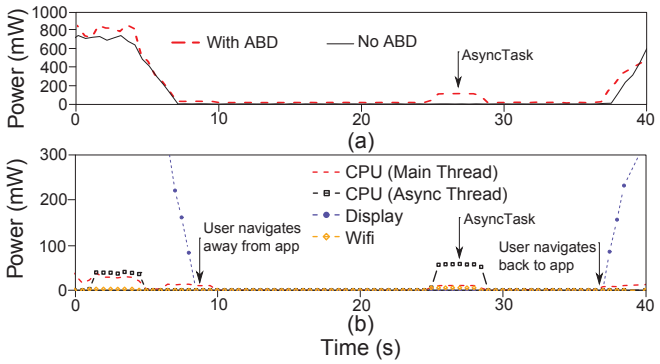

Fig. 10: Chronological traces of Boston Bus Map ((a) Total app power of similar traces with and without ABD manifested, (b) Component breakdown while ABD is manifested).

10(b) shows the power breakdown of each component of the trace with ABD manifested, showing an AsyncTask being executed while the user is not using the app. Knowing that the ABD could have been caused by the AsyncTask API (reported by the clustering analysis), the developer can quickly go the the identified region, which effectively reduce developer's attention from 2,007 lines to 213 lines of the app code.

D. Overhead Analysis

Power Overhead. We measure the power overhead of eDelta on a Nexus One Phone. The average power consumption of eDelta is 79.2 mW which accounts for 5.3% of the total power. The power overhead is caused by both the utilization tracker and the action tracker. eDelta's app instrumentation adds a small library, and typically 12 lines of byte code (13 if additional Dalvik registers are needed) to each instrumented API call. This increases app binary size by 4.32% on average, which is negligible.

Performance Overhead. Mobile apps are built atop the event-driven execution model to achieve interactivity. User interactions are translated to application events. Each event is registered with an event handler that is executed when the event is triggered. Event latency is highly correlated with user perceived performance. Thus, event latency reported by Android framework is adopted as the performance metric. Event latency of the instrumented version and original version of apps in Table IV is measured. The average performance overhead is 11.79%, which is moderate. Average event latency of apps in Table IV is less than 16.23ms. According to [22], users will not perceive a delay when the event latency is less than 100ms while interacting with the app. Thus, the instrumented version does not impact usage experience.

E. Discussion

eDelta pinpoints the APIs with high energy deviation. The reported APIs can usually provide helpful information for developers to correct the ABDs that manifest under certain contexts (e.g., particular user inputs/configurations). The ABDs that always manifest can be easily noticed by developers. APIs with high energy deviation will not be detected under the following scenarios: 1) ABD never manifests in all the collected traces, 2) ABD is triggered before the trace starts and remains throughout the entire trace for all the collected traces. There exists no energy deviation among the traces under these two scenarios. Thus, eDelta will not report APIs with high energy deviation to developers. Since eDelta collects traces from different users under various circumstances (e.g., particular user inputs/configurations), those scenarios did not happen in our experiments.

V. Conclusion

In this work, we have presented eDelta, a framework that assists app developers in pinpointing APIs with high energy deviation, which usually have high probability to be relevant to the abnormal battery drain. We have prototyped eDelta in Android 4.4 and have evaluated it with different real-world apps. Our results show that eDelta can effectively pinpoint the APIs with high energy deviation which cause ABD. Specifically, it reduces, on average, 94.6% of the amount of code that the developers would need to search to fix the ABD problem.

References

[1] A. Pathak *et al.*, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *HotNets*, Nov. 2011.

[2] Gigaom, "A demographic and business model analysis of today's app developer," http://research.gigaom.com/report/.

[3] X. Ma *et al.*, "eDoctor: automatically diagnosing abnormal battery drain issues on smartphones," in *NSDI*, 2013.

[4] Apigee, "Users Reveal Top Frustrations That Lead to Bad Mobile App Reviews," http://apigee.com/about/press-release, 2014.

[5] "K9Mail - Issue 3348 - K9 running CPU and data constantly." [Online]. Available: http://code.google.com/p/k9mail/issues/detail?id=3348

[6] A. Pathak *et al.*, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *MobiSys*, Jun. 2012.

[7] A. J. Oliner *et al.*, "Carat: Collaborative energy debugging for mobile devices," in *HotDep*, Oct. 2012.

[8] A. Pathak *et al.*, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof," in *EuroSys*, Apr. 2012.

[9] R. Mittal *et al.*, "Empowering developers to estimate app energy consumption," in *Mobicom*, Aug. 2012.

[10] L. Zhang *et al.*, "ADEL: an automatic detector of energy leaks for smartphone applications," in *CODES+ISSS*, Oct. 2012.

[11] P. Vekris *et al.*, "Towards verifying android apps for the absence of no-sleep energy bugs," in *HotPower*, Oct. 2012.

[12] C. Yoon *et al.*, "AppScope: application energy metering framework for android smartphones using kernel activity monitoring," in *USENIX ATC*, Jun. 2012.

[13] L. Zhang *et al.*, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *CODES+ISSS*, Oct. 2010.

[14] N. Balasubramanian *et al.*, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *IMC*, Nov. 2009.

[15] A. Pathak *et al.*, "Fine-grained power modeling for smartphones using system call tracing," in *EuroSys*, Apr. 2011.

[16] A. Shye *et al.*, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *Micro*, Dec. 2009.

[17] Y. Liu *et al.*, "Understanding and detecting wake lock misuses for android applications." in *FSE*, Jun. 2016.

[18] H. S. Park. *et al.*, "A K-means-like Algorithm for K-medoids Clustering and Its Performance." in *ICCIE*, 2006.

[19] R. J. Campello *et al.*, "A fuzzy extension of the silhouette width criterion for cluster analysis." *Fuzzy Sets and Systems*, 1987.

[20] "R," https://www.r-project.org/.

[21] "eDelta Technique Report," https://www.dropbox.com/sh/zj93v1buln227h9/AAC-x9BL_bm5y_nXCa1doqcpa?dl=0.

[22] Y. Zhu *et al.*, "Event-based scheduling for energy-efficient QoS(eQoS) in mobile Web applications." in *HPCA*, 2014.