2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) | 978-1-7281-7002-2/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICDCS47774.2020.00046

EnergyDx: Diagnosing Energy Anomaly in Mobile Apps by Identifying the Manifestation Point

Li Li[1], Xiaorui Wang[2], and Feng Qin[2]

[1]ShenZhen Institutes of Advanced Technology, Chinese Academy of Sciences
[2]The Ohio State University

***Abstract*—Abnormal battery drain (ABD) can negatively impact the user experience of smartphone apps, by consuming an unnecessarily high amount of energy and causing short battery lifetime. Unfortunately, user reports on ABD are usually too vague for app developers to precisely know how and when the ABD manifests. Therefore, it is important to have a diagnostic tool that can help app developers identify the ABD manifestation point for root cause analysis.**

In this paper, we propose EnergyDx, an automated diagnosis framework that assists developers in pinpointing the functions that either directly lead to or commonly coincide with the manifestation of ABD. EnergyDx features a novel 5-step analysis algorithm to distinguish the real ABD manifestation point from the power transition points caused by normal phone usage. We have prototyped EnergyDx in Android and evaluated it with 40 different real-world apps for diagnosing ABD cases caused by various types of issues. Our results show that EnergyDx reduces, on average, 93% of the amount of code that the developers would need to search for the root causes of ABD.

I. Introduction

While the computational capacity of the smartphone has improved significantly in the past few years, the battery technique has not improved at the same pace. Thus, battery life time has become a critical bottleneck for smartphones and highly impacts the smartphone user experience.

Abnormal Battery Drain. One type of the software defects that have been reported to trouble a large number of users is abnormal battery drain (ABD) [1][2]. An ABD usually consumes an unnecessarily high amount of energy and causes undesired fast battery drain. ABD can be caused by different issues (e.g., inefficient design, misconfiguration) resulting in overusing system resources. The ABD problem not only impacts end users, but also hurts the reputation of an app through negative user reviews [3][4].

Although users can easily notice the symptom, they often cannot remember precisely when and how an ABD is triggered. Most of the time user reports only describe the phenomenon and may not be helpful for the developers to even reproduce the ABD in the development environment [5]. Without insightful information, it is hard for developers to understand and fix the ABD problem. *Thus, an automated tool is sorely needed for app developers to diagnose the root cause of ABD in their app code*.

This work was done when Li Li was a Ph.D. student at The Ohio State University. Li Li would like to acknowledge the support, in part, from the National Key R&D Program of China (No. 2019YFB2102100), Science and Technology Development Fund of Macao S.A.R (FDCT) under number 0015/2019/AKP, Guangdong Key R&D Project (No. 2020B010164003), Shenzhen Discipline Construction Project for Urban Computing and Data Intelligence.

Existing Approaches are Insufficient. Existing approaches of handling the ABD problems can be mainly divided into two categories. The first category adopts dynamic approaches [6], [7], [3], [8] that try to detect ABD based on user traces collected at program runtime. For instance, eDoctor [3] identifies which app causes the ABD problem by analyzing the system resource usage information recorded at runtime. Then it suggests the end users uninstall the identified app. Though these approaches can effectively help users increase their battery life time, *the app-level information is often too coarse-grained for developers to identify where causes the ABD problem inside the app code*. The second category adopts static approaches that rely on the analysis of the app source code without running the app [9], [10], [11], [12], [13], [14], [15], [16], [17]. For instance, static dataflow analysis [9] is adopted to check whether a wakelock is correctly acquired and released in certain code paths, in order to detect a special type of ABD. Though these solutions can provide developers fine-grained information, they are usually limited to a certain type of ABD (e.g., no-sleep). Thus, it is *hard for them to diagnose the ABD caused by other (possibly unknown) issues (e.g., misconfiguration, looping)*. Hence, it is highly desired to have a novel methodology that can help developers (instead of users) diagnose the root causes of ABDs caused by various (and sometimes unknown) issues.

Key Observation and Challenge. After receiving ABD reports, developers usually hope to know the context information about the ABD (e.g., how and when the ABD manifests) in order to reproduce and diagnose the problem. To that end, we observe that when an ABD manifests, the power consumption of the whole app typically transits from low (normal) to high (abnormal). The system events of the mobile OS that are always invoked around the manifestation points can provide developers insightful diagnostic information of the ABD. Based on our investigation results, these events often either directly trigger or are closely related to the ABD. For example, when a user misconfigures the K9 Mail app [18], the app starts to keep making connection with a remote server, which abnormally consumes high power. Hence, identifying the event of user configuration can provide key information for developers to diagnose this example ABD (see Section III-B for more details). Therefore, it is important to accurately pinpoint this manifestation point and the closely-relevant events in the user traces.

Unfortunately, it is non-trivial to identify the manifestation point in real-world cases, because normal phone usage could also cause similar transitions in power consumption. For example, taking a picture can immediately transit the phone power consumption to a higher level. Such a power transition point can look similar to the manifestation of a real ABD. In addition, the power consumption values can sometimes be deceiving. For example, some long-running services (e.g., location, notification) can be executed concurrently with other normal events, making those events appear to be consuming more power than normal. Thus, how to distinguish the real ABD manifestation point from normal power transition points is a critical challenge.

Our Contribution. In this paper, we propose EnergyDx, an automated trace-based diagnosis framework that assists app developers in pinpointing the system events that are always invoked around the manifestation of ABD. With the reported event information, developers can directly go to the corresponding code segments to fix the ABD problem.

EnergyDx focuses on a diagnostic methodology. After the traces are collected from different users under various contexts, EnergyDx starts the diagnosis that features a novel 5-step analysis algorithm. In particular, EnergyDx first employs a ranking and normalization approach to remove the transition points caused by normal usage. After that, we adopt outlier analysis to differentiate the transition points caused by different issues based on the amplitude of power variation. The events and the corresponding callbacks within a certain range of the selected manifestation points are then reported to developers in an order based on their probability of causing the ABD. Developers can then get the context information of the ABD and narrow down their attention to the code segments called immediately after those events, which greatly reduces the searching efforts.

Specifically, this paper makes the following contributions:

- We make a key observation that power consumption of an app often transits from normal to abnormal when an ABD begins to manifest, and the relevant events can usually provide useful information for developers to fix the ABD.
- Based on the observation, we propose EnergyDx, an automated diagnosis framework that assists app developers in diagnosing ABD by identifying the real manifestation point and the relevant events.
- We have prototyped EnergyDx in Android. Our results show that EnergyDx can successfully diagnose the ABD caused by different reasons and can reduce, on average, 93% of the amount of code that the developers would need to search for the root causes of ABD.

The rest of the paper is organized as follows. Sections II and III introduce the design details of EnergyDx. Section IV presents our evaluation results. Section V discusses the related work. Section VI concludes the paper.

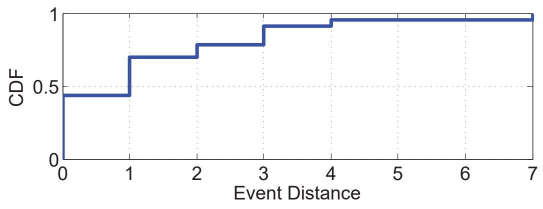


Fig. 1: Statistical analysis of event distance of 40 ABD cases.

II. Design of EnergyDx

In this section, we first introduce the key observation that motivates our design of EnergyDx. After that, we present the design overview of EnergyDx.

A. Key Observation

As background, in mobile OS like Android, the app logic is organized into activities. An activity contains widgets that users can interact with, such as buttons and text boxes. When users interact with widgets, Android OS and apps work together to dispatch the related events to the correct widgets. After that, the callbacks (e.g., onClick()) corresponding to certain widgets are invoked to perform related tasks. As a user interacts with an app, the activity can be in different states during its lifecycle. Thus, the events related to user interaction and activity lifecycle represent the main app logic.

We observe that *the events that are always invoked around the manifestation point of an ABD are often closely related with the ABD root cause.* ABD can be caused by complex events related to user interaction or activity lifecycle and their interactions. According to a report [19], many popular apps (e.g., K9Mail, Tinfoil and Wallabag) and even the Android framework sometimes failed to handle them correctly, resulting in abnormal battery drain. An ABD usually manifests after the occurrence of particular events in a particular code path. When an ABD is triggered, power consumption of the app commonly transits from normal (low) to abnormal (high).

To confirm our observation, we devise a metric named *event distance* to quantitatively analyze the relation between the ABD triggering event and the ABD manifestation point (i.e., the power transition point). Event distance is defined as the number of events (user interaction or activity lifecycle) invoked between (exclusive) the real triggering event (i.e., root cause) and the event that is closest to the manifestation point. The shorter the event distance is, the manifestation point is more closely related to the root cause.

We complete the statistical analysis of 40 real ABD cases caused by different issues. Figure 1 shows the analysis result. 90th percentile of the analyzed event distances is 3 or shorter. In an Android app, five events will typically be generated when a user simply switches from one activity (user interface) to another. Thus, the result shows that an ABD triggering event is indeed near the manifestation point in most of the cases. Manifestation points can lead developers to the events that are closely related to the ABD. Then, developers can directly go to the pinpointed code regions to fix the ABD problem.

```
1.Lcom/fsck/k9/activity/setup/AccountSettings; onResume
2.Lcom/fsck/k9/service/MailService; onCreate
3.Lcom/fsck/k9/activity/MessageList; onResume
4.Lcom/fsck/k9/K9Activity; onResume
5.Ljava/net/Socket;->connectV
```

Fig. 2: Events around the manifestation point (sample point 238 in Figure 3). The 1st event (line 1) is the root cause event. The 5th event (line 5) is the manifestation point, which may not always be logged in the trace.

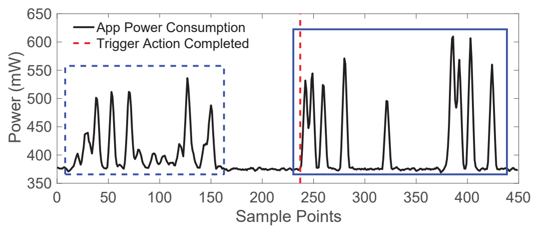


Fig. 3: Power trace of the K9 Mail ABD. The dotted square represents the normal usage. The solid square represents the power consumption when the ABD manifests.

Example. Figure 2 shows a trace of the K9 Mail app as an example. We record the events around the manifestation point when power consumption of the whole app transits from normal to abnormal (the ABD begins to manifest). The power transition is caused by a connection attempt with the remote server, as represented in the 5th line "*Ljava/fsck/Socket;->connectV*". The connection is triggered by user's misconfiguration of the account (see detailed discussion in Section III-B). As shown in Figure 2, after the user changes the account configuration ("*Lcom/fsck/k9/activity/setup/AccountSettings: onResume*" is invoked) and returns to the message list ("*Lcom/fsck/k9/activity/MessageList*"), the ABD begins to manifest. In this example, the root cause is the 1st event "*Lcom/fsck/k9/activity/setup/AccountSettings: onResume*". The manifestation point is the 5th event, so the event distance is 3 because there are 3 events between the two (exclusively). Sometimes, the manifestation event is *not* logged in the trace, because it is not related to either user interaction or activity lifecycle and thus not logged for the consideration of runtime logging overhead. In those cases, the logged event that is closest to the manifestation point is used instead. For example, if the 5th event is not logged in Figure 2, the 4th event would be identified as the manifestation point, which would still help quickly find the root cause (the 1st event).

Figure 3 plots the power consumption of the K9 Mail ABD. The x-axis is sampling time points. The y-axis is corresponding power consumption. Figure 3 not only shows that ABD manifestation causes power transition (around sample point 238), but also includes normal usage events that can also generate power transition points. These spikes (0 to 150 points, inside the dashed box) can generate misleading diagnosis information because they are the power consumption when

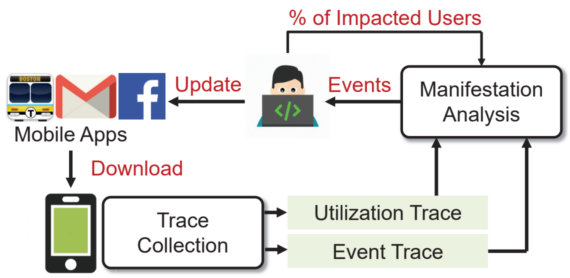


Fig. 4: The design and workflow of EnergyDx.

a user is composing an email. Thus, it is challenging to distinguish the ABD manifestation point from those transition points caused by normal usage.

B. Design Overview

EnergyDx aims to identify the ABD manifestation points in the collected traces. After that, EnergyDx reports to developers the events that are always invoked when the ABD begins to manifest. Though EnergyDx is not designed to directly identify the specific API that drains the battery, it can pinpoint the events closely related to the ABD. Developers can then check only the pinpointed code regions, so the search space for the root cause can be effectively reduced.

Figure 4 shows the system architecture and workflow of EnergyDx, which can be mainly divided into the following four parts. First, upon the report that a certain app causes abnormal battery drain, the developers can first instrument the app. It is important to note that, the developers are not required to manually instrument every event and just need to run the instrumenter (Section II-C) provided by EnergyDx. To reduce the runtime overhead caused by logging, EnergyDx only instruments the events related to user interaction and activity lifecycle. These events are not fine-grained, yet can sufficiently provide important diagnostic information. Second, when users download and run the instrumented app on their phones, *instrumented* events and utilization of system hardware components are logged into event traces and utilization traces, respectively. These traces are then transmitted to a remote server, when the smartphone is in charge with WiFi, which is a common practice to upload traces without impacting the normal usage of smartphone [3]. Thus, the transmission process does not impact the normal usage of smartphone. Third, after receiving the traces, the EnergyDx manifestation analysis (discussed in Section III) running on the backend server performs a novel 5-step analysis and catches the manifestation points in the collected traces. Finally, EnergyDx reports the events and corresponding callback functions that are always invoked when the ABD begins to manifest. It is important to note that the traces collected by EnergyDx are preprocessed to remove any user identifies, such as phone numbers or IP addresses in order to protect the user privacy.

Specifically, EnergyDx reports the events and their corresponding callbacks that coincide with the manifestation of

```
28223867 + Lcom/fsck/k9/service/MailService; onDestroy
28223867 - Lcom/fsck/k9/service/MailService; onDestroy
28224781 + Lcom/fsck/k9/activity/MessageList; onItemClick
28224844 - Lcom/fsck/k9/activity/MessageList; onItemClick
```

Fig. 5: A simple example of event log from K9 Mail.

energy anomaly to reduce developer's search space of the ABD root cause. Again, it is challenging to identify such a manifestation point because some normal operations may also cause power consumption increases. Second, it reports the power and corresponding event traces.

TABLE I: Examples of events related to user interaction and activity lifecycle.

Category	API Class Name	Example APIs
Activity Life Cycle Related	android.app.Activity	onCreate, onStart, onResume, onPause, onStop, etc.
UI Related	android.View	onClick, onLongClick, onKey, onTouch, etc.

C. Trace Collection

Two pieces of runtime information about the suspect app are recorded: Event durations and system resource utilizations, in order to estimate the online power consumption of each event.

Event Trace Collection. The event trace is collected through instrumenting events related to user interaction and activity lifecycle with the instrumenter provided by EnergyDx. We create a pool of the events to be instrumented. Table I shows the examples of common events in the created pool. The input of the instrumenter is the Android application package (APK), which contains all the Dalvik bytecode files and other resources (e.g., images) for an Android app. After receiving the APK file, EnergyDx first unpacks the APK file and disassembles the Dalvik byte code files into assembly-like format. Then it performs the instrumentation of the events related to user interaction and activity lifecycle. After that, it compiles the instrumented files back to Dalvik bytecode files and then packages them back to a new APK file. When users run the new APK files on smartphones, the durations of corresponding events are recorded using system timestamps. Figure 5 shows an example event trace, when a user uses the K9 Mail 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 of a certain event. Moreover, the name of the class in which the event is invoked is also recorded, such as *Lcom/fsck/k9/service/MailService*.

Utilization Trace Collection. In order to estimate the power consumption of the reported ABD app, we adopt the method proposed in [20] to implement a background service that periodically records the utilization of system components (i.e., CPU, display, WiFi, etc.). Specifically, it monitors the *proc* filesystem (procfs) to gather hardware utilization assigned to the target app. The utilization tracking is limited only to the suspect app identified by its PID due to the support for this functionality provided by the Linux kernel. Thus, the existence of multiple running apps does *not* affect utilization tracking of the suspect app. A tracking period of 500 ms is used by EnergyDx for a trade-off between power estimation accuracy and runtime logging overhead. Based on our experiments, a tracking period of 500 ms is sufficient to capture most energy anomalies because they need to last long enough to cause undesired battery drain.

III. Manifestation Point Identification

Manifestation analysis is the key feature of EnergyDx that distinguishes the real ABD manifestation point from the transition points caused by normal usage.

A. Manifestation Analysis Algorithm

Figure 6 shows the workflow of manifestation analysis, which mainly consists of five steps.

Step 1: Power Estimation of Events. The power consumption of each instrumented event (i.e., its corresponding callback function) is estimated based on the collected event durations and estimated app power consumption. Step 1 in Figure 6 shows the event power estimation process. Four example traces (e.g., E1 to E4) record the timestamps of starting and ending points of three example events (square, circle, triangle). Four power traces (e.g., P1 to P4) contain the timestamp of each sample point and the corresponding power consumption. Taking these traces as inputs, the power consumption of each of the three events is calculated by mapping each pair of power and event traces according to the timestamps. The traces collected from different users in Step 1 of Figure 6 are then represented as a sequence of events with their corresponding power in the chronological order.

Note that the power consumption estimated based on [20] is that of the entire app (not just that of a single thread) during the execution of this event. The estimation error is reported to be less than 2.5%, which is sufficient to characterize the app power transition. Power modeling is *not* a main contribution of our paper, so other more fine-grained techniques (e.g., [21]) can also be integrated with EnergyDx. In addition, to handle traces from phones with different hardware and software configurations, power model scaling [22] is performed to make their power data comparable.

Step 2: Event Ranking. Different events may have different power consumption, according to their functionalities. For example, retrieving data from a remote server consumes more power than simply processing user inputs. Transition points caused by raw power difference between different events can generate misleading information. Thus, looking at the raw power consumption of a single event instance is not correct. For a certain event, we analyze different instances across different traces and rank all the instances of the same particular event across all the traces based on their power consumption. As a result, the rankings can be used in the next step to normalize the power consumption of each event instance to its respective "typical" value, such that the normalized power is comparable among different events.

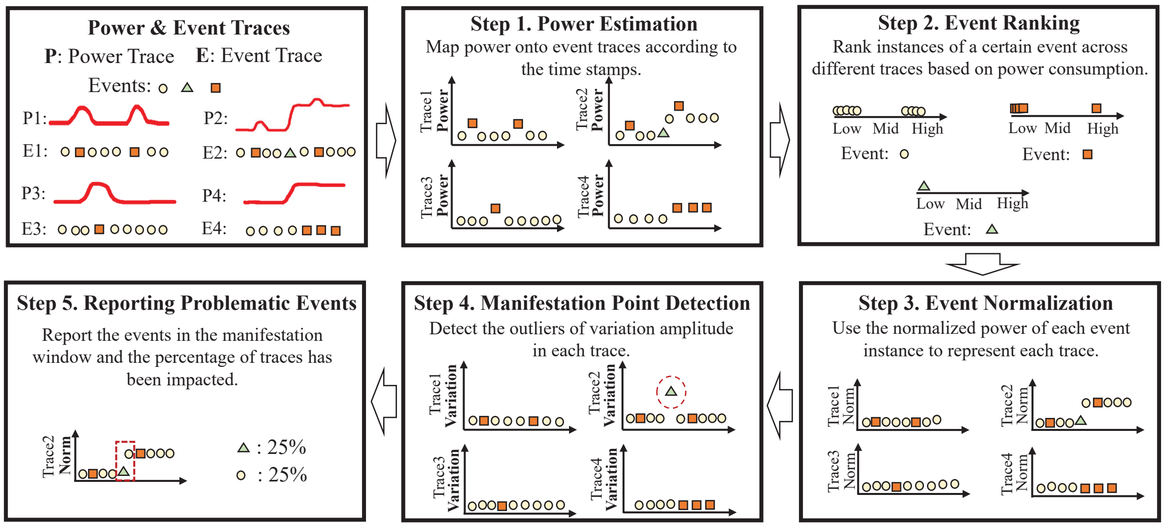

Power & Event Traces
P: Power Trace E: Event Trace
Events:
P1: P2: P3: P4:
E1: E2: E3: E4:
Step 1. Power Estimation
Map power onto event traces according to the time stamps.
Trace1 Power
Trace2 Power
Trace3 Power
Trace4 Power
Step 2. Event Ranking
Rank instances of a certain event across different traces based on power consumption.
Low Mid High
Event:
Step 3. Event Normalization
Use the normalized power of each event instance to represent each trace.
Trace1 Norm
Trace2 Norm
Trace3 Norm
Trace4 Norm
Step 4. Manifestation Point Detection
Detect the outliers of variation amplitude in each trace.
Trace1 Variation
Trace2 Variation
Trace3 Variation
Trace4 Variation
Step 5. Reporting Problematic Events
Report the events in the manifestation window and the percentage of traces has been impacted.
Trace2 Norm
: 25%
: 25%

Fig. 6: The work flow of EnergyDx manifestation analysis includes 5 steps to identify real ABD manifestation points.

Step 2 in Figure 6 shows the 3 events (in the 4 traces from Step 1) are ranked based on the analysis process. We can see from Step 1 that the events represented by squares consume higher power than those represented by circles during normal use due to functionality difference. Thus, the power transition points from circle events to square events in traces 1, 3 and 4 can generate misleading diagnostic information, because circle events and square events consume different amounts of power. However, after ranking, we see most of the instances represented by squares have similar ranking. Only the 7th instance of the square event from Trace 2 is ranked much higher than the other instances of the event. This event instance has a high probability to have been impacted by the ABD. In contrast to squares, the instances of the circle event are more evenly divided into two groups.

Step 3: Event Normalization. Based on the ranking information, we use an event normalization scheme to remove the transition points caused by the raw power difference between different events. Again, this is to compare the event power normalized to their respective "typical" value, such that *the power values are comparable among different events*. For a certain event, each instance is normalized to the power value at the 10th percentile of instances *across all the traces*. Specifically, given a trace, for any event instance i, on the axis, its normalized power p_{norm_i} is represented as $\frac{p_i}{p_{10thpercentile}}$, where $p_{10thpercentile}$ is the 10th percentile of the power consumption of other instances of the same event. Using the value at the 10th percentile as the base power is to reduce the impact of power outliers generated by previous steps (e.g., utilization tracking and power estimation). Moreover, the selection of power value at the 10th percentile gives us good experimental results, but this value can be adjusted for different training sets. The rationale for using this normalizing approach is that it eliminates the relative power consumption differences among different values, but keeps the difference among different instances of the same event. The instances that have relatively low normalized power (e.g., around 1, close to the base power) are invoked during normal usage. Those instances that maintain high normalized power have a high probability to have been impacted by the ABD.

After normalization, event instances in each trace are represented with normalized power (as shown in Step 3 of Figure 6). We see traces 1, 3 and 4 (Step 3 in Figure 6) are now flat and those transition points caused by power difference between various events are removed. For trace 2, event instances have low normalized power at the beginning. After the event represented by the triangle is invoked, the normalized power of the following instances increases to a high level, which indicates a high probability of being impacted by the ABD.

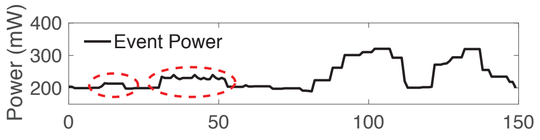
Step 4: Manifestation Point Detection. This step detects the ABD manifestation point based on the normalized power. When the ABD manifests, the normalized power consumption of the app transits from normal (low) to abnormal (high). Thus, event instances located at the manifestation point should show high power variation. In order to quantify the power variation caused by each event, we devise a metric named *variation amplitude*. The variation amplitude of the i_{th} instance V_i in a certain trace (based on the chronological order of all event instances) is calculated as $p_{norm\_i+1} - p_{norm\_i}$. Moreover, if the normalized power keeps increasing from the i_{th} instance until the $(i+n)_{\text{th}}$ instance, V_i is calculated as $p_{norm\_i+n} - p_{norm\_i}$. The intuition is that, in some real-world cases, the power consumption of the app gradually increases after the ABD is triggered and it will impact multiple events during the power increasing process. Step 4 in Figure 6 shows the variation amplitude attribution process of each event instance in the four traces. For traces 1, 3 and 4, normalized power remains almost flat and the corresponding variation amplitudes of those event instances remain at a low level. For trace 2, there exists high power variation at the event instance represented by the triangle as shown in Step 3. Thus, that instance is attributed with a much higher variation amplitude than the other instances.

Based on the variation amplitude of each event instance, we select the manifestation points by performing outlier detection. This is because most of the events keep their power flat and have low variation amplitudes during normal usage after event normalization. It is likely that only the event instances around the manifestation points have higher variation amplitudes. Specifically, for a certain trace, we first retrieve the variation amplitude of every event instance in the chronological order. After that, for the retrieved data set, we calculate the lower quartile $Q1$ (defined as the 25th percentiles) and the upper quartile $Q3$ (defined as the 75th percentiles). Then we obtain the interquartile range IQ defined as $Q3 - Q1$. The event instances whose variation amplitudes are larger than the upper outer fence (defined as $Q3 + 3 \times IQ$) are then selected as the possible manifestation points. The parameters of the algorithm (e.g., Q1, Q3) are decided through experiments which are determined to select out the manifestation points and to remove those noises generated by event power difference. As shown in Step 4 of Figure 6, no outlier is detected in traces 1, 3 and 4. For trace 2, one outlier, which is marked with a red dashed circle, is detected. The detected outlier is then treated as the point where the ABD begins to manifest.

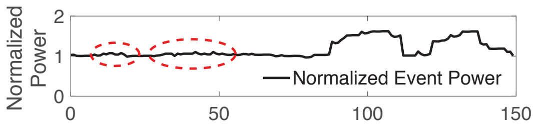
Step 5: Reporting Problematic Events. After the manifestation points are detected, we take all event instances within a certain range from the detected manifestation point (called the manifestation window) as potential events related to the ABD's manifestation. This is based on the observation made in Section II-A that the root cause of an ABD is normally near the manifestation point. In addition, having a manifestation window can result in two more benefits. First, it gives us more events to associate with the ABD since the user might need to do a few things in succession to make the ABD manifest. Second, the window also provides developers with more context information, such as what is happening when the ABD begins to manifest.

In order to provide developers more insightful diagnostic information, we sort the events within the manifestation window according to their relationship with the ABD, which is measured with the following scheme. Based on the feedback from online Android forum, through checking the numbers of app downloads and the user ABD reports [5], developers can normally estimate the percentage of users that have been impacted by the ABD. The developers could also hire a group of people to collect traces and find out this percentage with app-level detection tools, such as eDoctor [3]. The events that have impacted a similar percentage of users should have a higher probability to be related to the ABD manifestation. These events should thus be considered first.

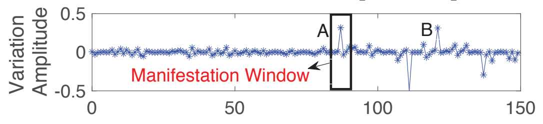
Thus, to sort events, we calculate the percentage of traces that a certain event in the manifestation window has impacted. Specifically, if one event is within the manifestation window of a certain trace, this trace is assumed to have been impacted by that event. Step 5 in Figure 6 shows the events in the manifestation window (assuming a size of 2 events for example) and the corresponding percentage of traces they have impacted. The events represented by the triangle have impacted one out of the four traces (i.e., 25% of the collected traces). As explained above, the traces are collected from different users under different contexts, thus events within the manifestation window can contain the ones that trigger the ABD and also those randomly generated during the users' normal interaction. This percentage-based sorting process can help filter out users' normal events, which often impacts a significantly different percentage of users from that provided by the developer (percentage of users impacted by the ABD).



(a) Raw Power Consumption of Events (Step 1)



(b) Normalized Power Consumption (Step 3)



(c) Variation Amplitude (Step 4)

Fig. 7: Diagnosis process of K9 Mail. The power transitions circled with dashes are not caused by the ABD.

B. A Real-World Example: K9 Mail

Figure 7 illustrates the diagnostic process with a real-world example: K9 Mail. Figure 7a plots the power consumption of each event instance in the chronological order. Note that the x-axis does not denote the exact execution time, but just discrete events in the chronological order (i.e., Step 1). The two parts circled with dashes represent the power transitions caused by raw power difference between different events. The events such as *Checkmail* consume more power than the others due to their different functionalities (e.g., refresh the mail list in this example). Thus, those power transitions are not real manifestation points and should be eliminated.

Figure 7b plots the normalized power of each event after Steps 2 and 3. As we can see, transition points caused by event power difference are removed by event power normalization. Figure 7c shows the variation amplitude of each event instance, calculated based on the normalized power (Step 4). Points A and B have high variation amplitudes due to high normalized power variation caused by them. Based on the variation amplitude of each event, transition point detection is then performed. Figure 8 shows the detection result. Points A and B in Figure 7c are detected as the manifestation points, because they have similar variation amplitudes which are much higher than the others in the trace. After that, we select the events within the manifestation windows of the two points (Step 5).

The above process is automatically repeated for all the collected K9 Mail traces. All the events that have impacted any traces are sorted based on how many traces they have impacted out of all the traces. Table II shows the first six events

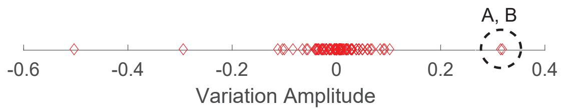


Fig. 8: Manifestation point detection process of K9 Mail.

TABLE II: Top K-9 Mail events reported by EnergyDx

Order, Event	%	**Order, Event**	%
1, AccoutSettings: onResume	16.6	**4**, AccountSettings:onCreate	8
2, MessageList:onResume	16.6	**5**, MailService:onDestroy	8
3, K9Activity:onResume	16.6	**6**, MailService:onCreate	25

whose percentages are closest to the percentage value provided by the developers (15% in this case). Among those events, *MailService* is an Android service that runs in the background to send and receive email. *MessageList* is an activity that lists all emails to the user. *AccountSettings* is an activity that allows the user to change preferences. Knowing that the energy anomaly is associated with these three events, the search space of the root cause can be effectively reduced from 98,532 lines to only 161 lines of the app source code.

We now look into the pinpointed 161 lines for the ABD root cause. By checking the event *AccountSettings: onResume*, which is always invoked when the ABD manifests, we find that the developers did not set a limit on the allowed number of connections to the IMAP server. For instance, Gmail only allows 15 simultaneous IMAP connections per account. As a result, a user can change this setting in the *AccountSettings* activity to a high number that exceeds the limit allowed on the IMAP server and thus gets declined by the server. This wrong setting makes the app periodically try to connect with the server, which results in the undesired ABD. The information reported by EnergyDx can indeed effectively help developers reduce the search space of finding this real root cause. This has been confirmed by the GitHub site of K9 Mail, which shows that the developers have found and fixed this ABD.

IV. Evaluation

We now evaluate EnergyDx with 40 downloaded apps. Table III shows the corresponding information of each app.

A. Experimental Methodology

We implement EnergyDx in Android 4.4. The manifestation analysis is implemented in Python and R [23]. Real-world phone usage and power traces are collected from more than 30 different volunteer users with various smartphones. It is important to note that EnergyDx does not depend on any features particular to Android 4.4 and so has no problem running on the newest version of Android. In order to evaluate the effectiveness of EnergyDx, we apply EnergyDx to 40 different ABD apps. The app selection is mainly based on the following rules. First, some of the 40 apps (e.g., Facebook, K-9 Mail) have been studied by other papers [3], [12]. We select these apps to show EnergyDx works on these well known cases.

TABLE III: Apps used to evaluate EnergyDx.

ID	**App**	**Downloads**	**Root Cause**	**Code**
1	Facebook	1B+	no-sleep	98.5%
2	Boston Bus Map	100k+	loop	86.04%
3	K-9 Mail	5M+	configuration	99%
4	CommonsWare	10M+	no-sleep	85.2%
5	Open Camera	10M+	no-sleep	98.3%
6	Droid VNC	1M+	no-sleep	94.46%
7	Binaural-Beats	5M+	no-sleep	95.6%
8	Zmanim	100K+	no-sleep	96.5%
9	MonTransit	500K+	no-sleep	94.1%
10	Aripuca	100K+	no-sleep	96.2%
11	Conversations	10K+	configuration	96.6%
12	Ushahidi	50K+	no-sleep	91.6%
13	Sofia Navigation	50K+	configuration	96.5%
14	Osmdroid	5K+	no-sleep	87.3%
15	Geohashdroid	n/a	no-sleep	96.2%
16	BabbleSink	50K+	no-sleep	82.4%
17	Traccar	50K+	no-sleep	96.2%
18	Tinfoil	n/a	loop	92.4%
19	Pedometer	100K+	configuration	91.7%
20	FBReader	500K+	no-sleep	90.1%
21	Owncloud	100K+	configuration	97.3%
22	Sensorium	50M+	no-sleep	92.1%
23	Signal	500K+	loop	98.3%
24	Summit APK	500+	no-sleep	89%
25	ValenBisi	10M+	no-sleep	93.5%
26	Ulogger	n/a	no-sleep	85.7%
27	AAT	50K+	no-sleep	97.4%
28	Wallabag	1M+	configuration	98.57%
29	Tomahawk Player	n/a	no-sleep	89.9%
30	Call Meter	n/a	no-sleep	96.69%
31	Simple Note	50K+	configuration	98.8%
32	NextCloud	50K+	configuration	99.3%
33	ArtWatch	5M+	loop	92.3%
34	WADB	1M+	no-sleep	94.3%
35	MFacebook	500K+	loop	99%
36	Kryptonite	500+	no-sleep	97.2%
37	Flybsca	10K+	configuration	96.6%
38	Throughput	n/a	loop	98.3%
39	Piano	n/a	no-sleep	98.3%
40	Fitdice	n/a	configuration	93.7%

Second, other apps (e.g., NextCloud, MFacebook) are selected because they are reported in online Android forums to have abnormal battery drain. Also, we select those apps because they cover the most common ABD root causes: configuration, no-sleep and loop. Configuration ABD represents the case in which an app is misconfigured leading to fast battery drain. No sleep ABD does not correctly release certain resources. Loop ABD triggers the app to periodically perform unnecessarily tasks. According to the investigation in [2], the three issues account for about 89.3% of all the causes.

B. Overall Results

The metric *code reduction* is used here to evaluate the effectiveness of EnergyDx and is defined as $\frac{N_{All}-N_{Diagnosis}}{N_{All}}$, where $N_{Diagnosis}$ represents the number of code lines responsible for the events that EnergyDx reports for developer to diagnose, and N_{All} represents the entire code lines of the app. Intuitively, the higher the code reduction is, the more useful the diagnostic information provided by EnergyDx is. The overall result shows that EnergyDx reduces, on average, 93% of the amount of codes that developers need to search for the root causes of ABD. Based on the fine-grained diagnostic information provided by EnergyDx, we have been able to fix the ABDs of all the 40 apps and got confirmed with the developers, either by finding the same fixes in their newer versions or getting acknowledged through communication with them.

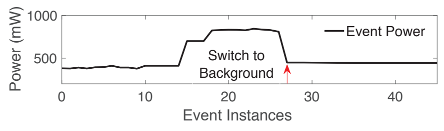
Comparison with Existing Approaches. As discussed before, most existing approaches either try to detect which app causes ABD or analyze the app source code to find only a specific type of ABD (e.g., no-sleep). Thus, they focus on different problems and are thus not comparable with EnergyDx. However, in order to highlight the contributions of EnergyDx, here we try to compare with two best-related state-of-the-art studies: No-sleep Detection [9] and eDelta [10]. We select No-sleep Detection as a baseline because they report that many ABDs are caused by no-sleep bugs (e.g., wakelock or sensors are not properly released) [9]. Of course, because it is designed to find only no-sleep ABD by leveraging data flow analysis in the source code, it cannot detect other types of ABD. The second baseline, eDelta [10], is designed to detect high energy deviation APIs in the app code by assuming that the energy consumption of some APIs would rise above a certain threshold after ABD manifestation. However, when the ABD is caused by an API whose energy deviation is relatively small (but might last long), eDelta is not able to detect it.

We compare EnergyDx with No-sleep Detection and eDelta by applying all the three approaches to the 40 tested apps. Note that both No-sleep Detection and eDelta are designed as detection tools that are supposed to identify the root cause. Hence, if they cannot detect the right root cause, these two approaches cannot reduce the search space of the right root cause. In that case, their code reduction would be 0%. In sharp contrast, EnergyDx is a diagnosis tool that aims to provide developers a list of possible events impacted by the ABD for the developers to find the real root causes themselves. In our results, No-sleep Detection is able to detect the root causes of all the 21 apps that indeed have no-sleep ABD (i.e., 100% code reduction). But for the other 19 apps without no-sleep ABD, No-sleep Detection cannot provide any useful information and thus has a 0% code reduction. Overall, its code reduction is $(100\% \times 21 + 0\% \times 19)/40 = 52.5\%$. Similarly, eDelta is able to detect the ABD of 26 apps that have high-energy deviation APIs, but has a 0% code reduction for the rest apps. Overall, eDelta's code reduction is 65%. Both are much lower than that of EnergyDx (93%). Again, both baselines are not exactly designed for ABD diagnosis. This comparison is just to highlight the importance of ABD diagnosis.

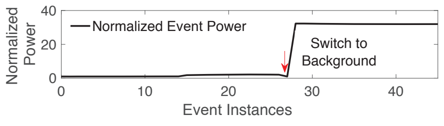
C. Case Study with Typical Apps

In this section, we present detailed analysis when EnergyDx is applied to apps with different ABD issues.

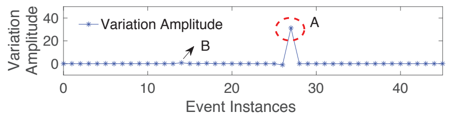
OpenGPS is an open-source app used to track a user's location [24]. Figure 9 shows the diagnostic process. Figures 9a, 9b, and 9c plot the raw event power, normalized event power, and variation amplitude of each corresponding event as a user interacts with the app. Figure 10 shows the transition



(a) Raw Power



(b) Normalized Power



(c) Variation Amplitude

Fig. 9: Manifestation point identification for OpenGPS.

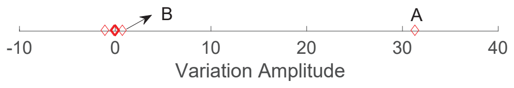
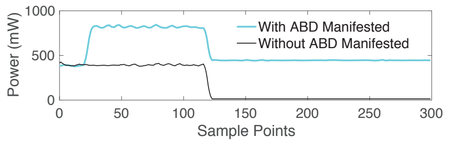


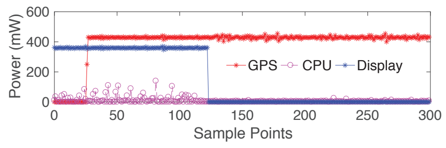
Fig. 10: Manifestation point identification for OpenGPS.

points detection process according to the variation amplitude of each event instance in Figure 9c. Two transition points are detected (e.g., points A and B as shown in Figure 9c) in this case. Those events within the manifestation windows (e.g., around A and B) are then selected. After that, we calculate the percentage of users that have been impacted by each selected event. According to the information (e.g., percentage of users impacted by the ABD) reported by the developers, we sort different selected events based on how close their percentages are to the reported one.

After sorting the selected events, the first two events are *LoggerMap:onPause()* and *Idle(No_display)* (Table IV). *LoggerMap:onPause()* is an activity-lifecycle event that is invoked when the current activity is switched to the background.



(a) Power consumption of the whole app.

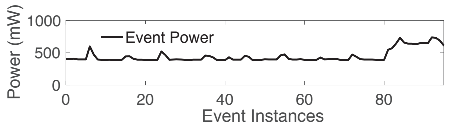


(b) Power breakdown when the ABD manifests.

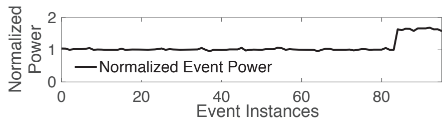
Fig. 11: Power breakdown of OpenGPS when the ABD manifests. GPS keeps consuming power in the background.

TABLE IV: Events reported to developers and their corresponding order (based on percentage) of the OpenGPS app.

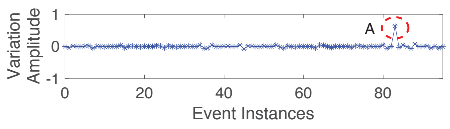
Order, Event	Order, Event
1, [LoggerMap: onPause()]	3, [LoggerMap:onResume()]
2, [Idle(No_Display)]	4, [ControlTracking:onPause()]



(a) Raw Power.



(b) Normalized Power.



(c) Variation Amplitude.

Fig. 12: Diagnosis process of Wallabag.

Idle(No_display) is the event invoked when the app is in the background. The two reported events show that the ABD manifests when the *LoggerMap* activity is switched to the background. With that diagnostic insight, the developers can directly go to the pinpointed code region. After further investigating the source code, we find that the location service is not correctly released in the *LoggerMap* activity. To illustrate the root cause of the ABD, Figure 11 shows the power breakdown of the app when the ABD manifests. The GPS keeps consuming power even after the app is switched to the background (e.g., display power is 0) without rendering any information to the user. By using EnergyDx, the search space is reduced from 5,060 to 569 lines of code, which effectively reduces developers' effort of fixing the ABD problem.

Wallabag is an open-source app [25] that allows users to read their articles on both the mobile and web sites. Figures 12a, 12b, and 12c show the raw event power, normalized event power, and variation amplitude of an example trace of Wallabag, respectively. Figure 12b shows that there is a large increase of the normalized event power (e.g., at the point circled with dashes in Figure 12c). After that, the app power consumption transits from normal (low) to abnormal (high) and keeps at a higher level. Figure 13 illustrates the manifestation point identification process. Point A (shown in Figure 12c) is selected out as the manifestation point. Events within the manifestation window (e.g., around A) are then reported to the developers.

Table V lists the reported events. The first three events are *ReadArticle:menuDeleted*, *ReadArticle:onCreate* and *ReadArticle:onResume*. These three events are always invoked when

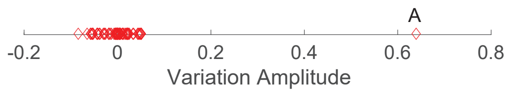
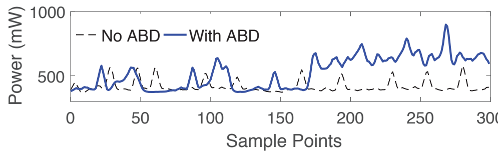
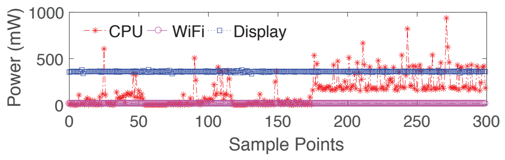


Fig. 13: Manifestation point identification for Wallabag.



(a) Power consumption of the whole app.



(b) Power breakdown when the ABD manifests.

Fig. 14: Power breakdown of Wallabag.

the ABD begins to manifest. *ReadArticle* is an activity that allows a user to save and read their articles later. *menuDeleted* is a button designed for users to delete the selected article. Figure 14 shows the power breakdown when the ABD manifests. As can be seen, the app consumes high CPU power when the ABD begins to manifest. With the provided information, developers can directly go to the pinpointed code segments.

After further investigating the code, we find that when a user tries to delete an article on the mobile client (i.e., event menuDeleted) that has already been deleted on the server side. The app keeps retrying to sync with the server, which causes high battery drain. This is the reason why those three events are always invoked when the ABD begins to manifest. With the provided information, developer's search space is reduced from 21,424 lines to 306 lines of code.

Tinfoil is a social communication app [26]. The variation amplitude attributed to each event is sent to the manifestation point identification process. Figure 15 shows the diagnosis result. Table VI lists the four reported events. The first two are *Idle(No_Display)* and *FbWrapper:menu_item_newsfeed*.

TABLE V: Events reported to developers and their corresponding order (based on percentage) of the Wallabag app.

Order, Event	Order, Event
1, [ReadArticle:menuDeleted]	4, [LibsActivity:onCreate()]
2, [ReadArticle:onCreate]	5, [BaseActionBarActivity:onCreate]
3, [ReadArticle: onResume]	6, [LibsActivity:onResume]

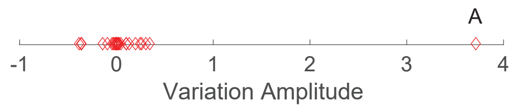


Fig. 15: Manifestation point identification for Tinfoil.

TABLE VI: Events reported to developers and their corresponding order (based on percentage) of the Tinfoil app.

Order, Event	Order, Event
1, [FBWrapper:menu_item_newsfeed]	3, [FBWrapper:menu_about]
2, [Idle(No_Display)]	4, [Preferences:onResume]

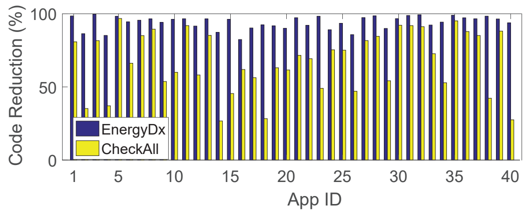


Fig. 16: Code reduction compared with the baseline that reports all the transition points.

Among them, *Idle(No_Display)* is the event that indicates the app is in the background. *FBWrapper:menu_item_newsfeed* is the event that navigates a user to the interface of news updating. After investigating the source code in the pointed segment, we find that in the news updating interface (triggered by the event *FBWrapper:menu_item_newsfeed*), the app keeps communicating with the remote server and retrieves the new information to render it on the interface for users. However, when the app is switched to the background, the app still keeps syncing with the server to render an invisible interface, which causes the ABD. This information helps reduce the search space from 4,226 lines to 236 code lines.

D. Comparison with Checking All Transition Points

In this section, we compare EnergyDx with a baseline named *CheckAll*. The metric *code reduction* defined in Section IV-B is adopted for the comparison. The baseline CheckAll performs Step 1 of EnergyDx to estimate the power consumption of each event. Different from EnergyDx, CheckAll does not try to distinguish the real ABD manifestation point from normal power transition points. Instead, CheckAll reports all the events that are invoked around all the power transition points to the developers. This comparison is to highlight the importance of identifying real ABD manifestation point.

Figure 16 shows the code reduction of the 40 apps with the two different schemes. The x-axis is the id of each app. For example, app 3 is the K9 Mail app; app 28 is Wallabag; app 18 is Tinfoil. On average, developers only need to search 168 lines of code with EnergyDx (code reduction 93%) for the ABD root causes, but 1,205 lines with CheckAll (code reduction 67%). For instance, for K9 Mail (app 3), with the information reported by EnergyDx, the developer needs to check only 161 lines. In contrast, with the information reported with CheckAll, the developer would need to check 9,845 (61 times) lines of code. The reason is that through event normalization, EnergyDx removes the transition points caused by raw power difference between different events. Moreover, through transition point differentiation, EnergyDx reports only the events that are closely related to the ABD manifestation.

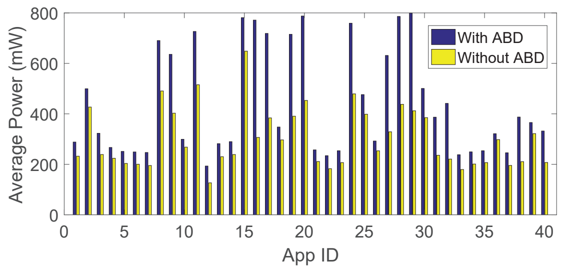


Fig. 17: Average power comparison of each app before and after the ABD is fixed.

E. App Power Consumption Reduction

In this section, we compare the average power consumption of each app before and after the ABD is fixed according to the diagnostic information provided by EnergyDx. Figure 17 shows the corresponding result. The average app power consumption has reduced by 27.2% after the ABD is fixed. The decreasing percentage for different apps varies for the reason that the ABD cases are caused by different issues which overuse various hardware components and consume different amount of power (e.g., GPS, CPU).

F. System Overheads

In order to not impact the smartphone users, the apps instrumented by EnergyDx should not have a much longer latency or much higher power consumption. Note that the manifestation analysis steps discussed in Section III are conducted on a remote server and so their overheads do not affect the users.

Performance Overhead. According to the investigation in [27], users will not perceive a delay when the event latency is less than 100ms, during the process that a user interacts with an app. Thus, event latency reported by the Android framework is adopted here as the performance metric. For each tested app, we measure the event latency difference between 1) the original version and 2) the version instrumented by EnergyDx. The average latency increase is 8.3%. Moreover, the average event latency of all the instrumented apps is less than 9.38ms. Thus, the performance overhead is moderate.

Power Overhead. We measure the power consumption of EnergyDx on a Nexus 6 smartphone using a Monsoon Power Monitor. The average power consumption of EnergyDx is 32mW. This accounts for only 4.5% of the total power of smartphone during the usage process, which is moderate. The power overhead is mainly caused by: 1) the utilization information collection and 2) the event information collection.

V. Related Work

Our work is closely related to *energy bug detection*. There has been a considerable amount of research in app bug detection and diagnosis [7], [6], [3], [28], [12], [13], [14], [15], [29], [16], [17], [30]. Existing research can be mainly divided into two categories. The first category detects which app causes ABD for end users [6], [7], [3], [8], while the second category detects the APIs that have been misused in the source code for app developers [9], [10], [11], [12], [13], [14], [15], [16], [17].

In the first category, eDoctor [3] clusters the resource usage information during the app execution into different phases and detects the abnormal app based on the clustering result. Oliner et al. [6] design a collaborative approach to detect which app has energy bugs and energy hogs. Though those approaches can efficiently help end users differentiate the ABD app, the reported app-level information is often too coarse-grained for developers to pinpoint the root cause in the app code. In contrast, EnergyDx provides detailed diagnostic information to help developers reduce the search scope of the ABD root cause.

In the second category, Pathak et al. [9] use data flow analysis to detect whether a wakelock is acquired but not correctly released in a certain code path. The solution is designed for a particular type of ABD. GreenDroid [12], [13] diagnoses energy problems in Android apps by detecting whether corresponding sensors are not timely and correctly deactivated. Though these solutions can provide developers fine-grained information, they are usually limited to a certain type of ABD (e.g., no-sleep). In contrast, EnergyDx can diagnose ABD caused by various (and even unknown) issues.

eDelta [10] is designed to detect high energy deviation APIs in the app code and requires fine-grained API instrumentation. However, when the ABD is caused by an API whose energy deviation is relatively small (but might last long) or an API that is not instrumented, eDelta fails to detect the ABD in these scenarios.

VI. Conclusion

In this work, we have presented EnergyDx, an automated diagnosis framework that assists app developers in pinpointing the root cause of an ABD in the app code. EnergyDx provides developers diagnosis information by identifying the ABD manifestation point and reporting the events around the selected point. These events can provide developers context information (how and when the ABD manifests) about the ABD and reduce the search space for the root cause. We have prototyped EnergyDx in Android and evaluated it with 40 different real-world apps. Our results show that EnergyDx reduces, on average, 93% of the amount of code that the developers need to search for the ABD root cause.

References

[1] Gigaom, "A demographic and business model analysis of today's app developer." https://research.gigaom.com, 2013.

[2] A. Pathak, Y. C. Hu, and M. Zhang, "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.

[3] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "eDoctor: automatically diagnosing abnormal battery drain issues on smartphones," in *Proceedings of 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

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

[5] K-9 Dog Walkers, "K9Mail - Issue 3348 - K9 running CPU and data constantly," 2013. [Online]. Available: http://code.google.com/p/k9mail/issues/detail?id=3348

[6] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stocia, "Carat: Collaborative energy debugging for mobile devices," in *Proceedings of the 8th USENIX conference on Hot Topics in System Dependability*, Oct. 2012.

[7] H. Kim *et al.*, "Detecting energy-greedy anomalies and mobile malware variants," in *MobiSys*, Jun. 2008.

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

[9] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "Proceedings of the 10th international conference on Mobile Systems, applications, and services," in *What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps*, Jun. 2012.

[10] L. Li, B. Beitman, M. Zheng, X. Wang, and F. Qin, "eDelta: Pinpointing Energy Deviations in Smartphone Apps via Comparative Trace Analysis." in *Proceedings of 8th International Green and Sustainable Computing Conference*, 2017.

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

[12] Y. Liu *et al.*, "GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications." *IEEE Transactions on Software Engineering*, 2014.

[13] J. Wang *et al.*, "E-greenDroid: effective energy inefficiency analysis for android applications." in *Proceedings of the 8th Asia-Pacific Symposium on Internetware*, 2016.

[14] T. Wu *et al.*, "Relda2: An effective static analysis tool for resource leak detection in Android apps." in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[15] ——, "Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps." *IEEE Transactions on Software Engineering*, 2016.

[16] Y. Liu *et al.*, "Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications," in *IEEE Software*, 2015.

[17] ——, "Understanding and detecting wake lock misuses for Android applications," in *24th ACM SIGSOFT International Symposium on Foundations of Software Enginnering*, 2016.

[18] K-9 Dog Walkers, "K9Mail: An advanced Email client for Android," 2018. [Online]. Available: http://code.google.com/p/k9mail/

[19] B. Abhijeet, C. Lee Kee, C. Sudipta, and R. Abhik, "Detecting energy bugs and hotspots in mobile apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[20] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and Y. Lei, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[21] A. Pathak *et al.*, "Where is the Energy Spent Inside My App? ." in *the 7th ACM European Conference on Computer System*, 2012.

[22] M. Radhika, K. Aman, and C. Ranveer, "Empowering developers to estimate app energy consumption," in *Proceedings of the 18th annual international conference on Mobile Computing and Networking*, 2012.

[23] R, "The R Project for Statistical Computing," https://www.r-project.org/, 2018.

[24] Rene de Groot, "OpenGPSTracker." https://github.com/rcgroot/open-gpstracker, 2017.

[25] Wallabag, "A self hostable application for saving web pages," https://github.com/wallabag/android-app, 2017.

[26] Daniel Velazco, "Tinfoil." https://github.com/velazcod/Tinfoil, 2017.

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

[28] L. Ravindranath *et al.*, "Appinsight: Mobile app performance monitoring in the wild," in *OSDI*, 2013.

[29] M. Wan *et al.*, "Detecting Display Energy Hotspots in Android Apps." in *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015.

[30] M. Hoque *et al.*, "Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices." *ACM Computing Surveys*, 2016.