A Comparison of Energy Bugs for Smartphone Platforms

Jack Zhang, Ayemi Musa and Wei Le
Rochester Institute of Technology
One Lomb Memorial Drive, Rochester, NY, USA
{jxz8072, ajm6611, wei.le}@rit.edu

Abstract—**Bugs that cause reduced battery life in smartphones have been classified as energy bugs. The majority of previous research on energy bugs focused on the Android platform. The goal of this work is to identify the similarities and differences of energy bugs for the Android, iOS and Windows platforms, based on which, we can then determine whether the techniques of managing energy bugs developed for one platform can be potentially applicable in other platforms. Our results show that applications in Android and Windows share similar root causes for leading to battery drain, but energy bugs in iOS applications are produced differently. Our conclusions are drawn based on a comparison of power models for the three platforms and an analysis of the bugs from 6 applications common to Android, iOS and Windows.**

I. Introduction

Smartphones have been an integral part of our daily activities. An important factor that determines whether a smartphone is trustable for long running tasks, such as health monitoring or navigation, is its battery life. Due to the importance, there have been much work [1]–[4] on analyzing behaviors and root causes of energy bugs that can lead to battery drain. The majority of such work focus on Android applications, assuming the techniques developed for Android are generalizable to different smartphone platforms. Although intuitive, we believe that further evidences need to be drawn for such generalization, as the design philosophies of Java (for Android), Objective-C (for iOS) and .NET (for Windows) are different from one another. For example, Pathak et. al [2] has conducted studies on non-sleep energy bugs with the emphasis on the use of the wakelock APIs. These APIs are Android specific; that is, no guarantee that the same wakelock APIs would appear on other smartphone platforms. Thus, the question remains whether other smartphone platforms potentially have the same types of non-sleep energy bugs and whether the solutions for non-sleep bugs would be generally applicable.

The goal of this paper is to not only complement previous efforts on understanding and characterizing energy bugs, but also establish a basis for potentially generalizing energy bugs across platforms. Specifically, we aim to answer the following two questions:

1) Do the Android, iOS and Windows platforms use similar APIs for power management?
2) Do energy bugs of the three smartphone platforms have the same root causes?

Our research takes two steps. First, for each platform, we identify APIs related to power management and understand how they are used in the applications, based on which, we construct power models for the Android, iOS and Windows platforms. Second, we analyze energy bugs from the three platforms. The focus is to identify the causes of these bugs and coding patterns related to the bugs. For comparison, we first collect 6 real-life energy bugs from important Android applications such as Facebook and GoogleMaps. For each Android bug, we search similar bugs for both the Windows and iOS platforms occurred in the same application and compare their root causes. It should be noted that here, we only focus on battery drain caused by application level bugs rather than problems in hardware or OS.

Our results show that although the architecture and the power models are different for Android and Windows, there is a similarity on how an API can give the programmer the ability to control the power usage for the applications. Because of that, there are types of energy bugs on Android and Windows that share root causes. iOS, on the other hand, implements a different power model and thus the root cause for the energy bugs is different from iOS and Windows.

In summary, this paper performs a preliminary study on characterizing and comparing energy bugs across the Android, Windows and iOS platforms. Our contributions include:

- identification and comparison of power models for the three smartphone platforms;
- analysis and generalization of the energy bugs based on their root causes.

The paper is organized as follows. In Section II, we present a comparison of power models for Android, Windows and iOS. In Section III, we explain the collection and analysis of the energy bugs. In Section IV, we present the related work, followed by a discussion in Section V and the conclusions in Section VI.

II. Power Models for Android, Windows and iOS

A *power model* describes the behavior of application with respect to energy consumption in mobile phones. Incorrectly understanding and implementing these models can lead to excessive use of battery or battery drain. From another perspective, comparing the power models can help understand the root causes of energy bugs across different platforms. Here,

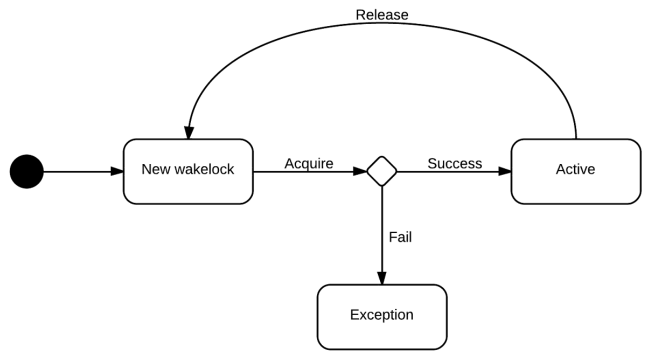

Fig. 1: Android Power Model

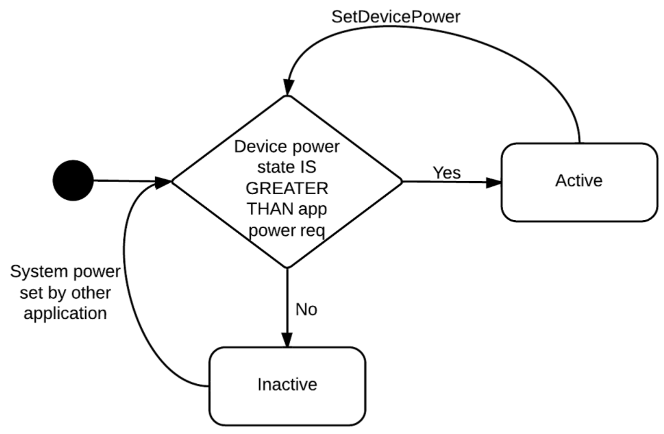
SetDevicePower
Device power state IS GREATER THAN app power req
Yes
Active
No
Inactive
System power set by other application

Fig. 2: Windows Power Model

we present a comparison on power models for the Android, Windows and iOS platforms [5], [6], [7], [8]. We construct the power models by identifying for each platform, what power states exist for an application and in what scenarios these states can be changed. We then represent the states and their transitions using finite state machines. Comparing the three identified power models, we find that power management for Android and Windows are similar in that it is done directly through system API calls in the apps, while in iOS, the system manages the power for applications through notification (events) sent by the apps.

Android. Our initial study on Android power models focuses on the *wakelock* APIs. In Android apps, wakelock is used as a flag for apps to notify whether the CPU should stay awake or go to sleep; if awake, what power state the phone should be set at. The basic assumption here is that the phone consumes more power when it is awake than when it is asleep. In Figure 1, we show how an app typically uses a wakelock. To use a wakelock, the developer will first create a new wakelock with a parameter that indicates what the phone's power state should be at. Currently, Android supports four types of parameters for creating a wakelock, including *partial_wake_lock*, *full_wake_lock*, *screen_dim_wake_lock* and *screen_bright_wake_lock*. Based on the parameter, the application then acquires the wakelock. If successful, the application will remain in the active state until the release function is called. Otherwise, the application enters an exception state. The developers can choose to handle this exception by re-acquiring the wakelock. It should be noted that the Android system supports a reference counter for wakelocks. That is, we can acquire the same wakelock multiple times in the apps before releasing it.

Windows. In Windows, the power state is classified as *full on*, *low on*, *standby*, *sleep* and *off*. Developers will specify a minimum power requirement for each application. As shown in Figure 2, when an application starts to run, if the device's current power state is equal to or greater than the minimum power requirement the application has, the application transitions to the active state. If the device's current power state is less than the minimum requirement, the application enters the inactive state. From the active state, the application can start executing its own tasks. The application can also reset its own minimum power requirement using the API *SetPowerRequirement*, in which case, the application will go through the power requirement check again to see if it will stay in an active state. Similarly, at the active state, the application can reset the device's power state using the API *SetDevicePower*. This action has a global impact in that all the applications in the system would go through the power requirement checks for determining which states whey should be at [5], [6].

iOS. In iOS, the power usage depends on the application state, namely *foreground inactive*, *foreground active*, *background inactive* and *background active*, shown in Figure 3. The application only can run at one of the above states. When an application is launched, it immediately enters the *foreground inactive* state. This state is used for preparing for the *foreground active* state and the *background active* state. By default, the app will transition from the *foreground inactive* state to the *foreground active* state. The iOS only allows one *foreground active* application running a time. When the application at the *foreground active* state, the app will be displayed on the foreground and the user can interact with it directly. From the *foreground active* state, the user has the control to keep the app running in the *foreground active* state, or transition it to the *background active* state. For example, when a user clicks a home button on the iPhone, the current *foreground active* application will transition to the *background active* state. As shown in Figure 3, the transition takes two steps. The app will first enter the *foreground inactive* state to prepare for the transition and then enter the *background active* state.

The difference between the *background active* and *background inactive* states is that the *background active* state will still be executing code, while the *background inactive* state will have all its actions suspended. The OS will decide when an app should transition from the *background active* state to the *background inactive* state. Similarly, in the *background inactive* state, the OS will determine if the application needs

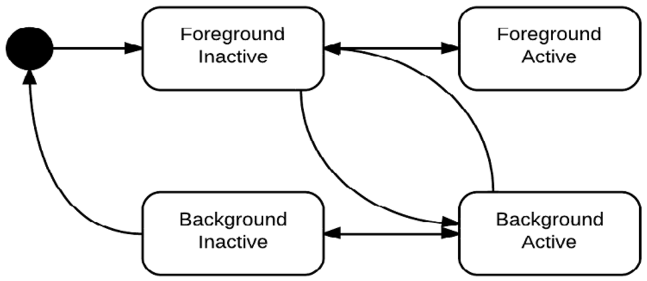

Fig. 3: iOS App State Graph

to be terminated completely. In either the *background inactive* or *background active* state, the user can put the app back to the *foreground active* state. The arrows in this power model do not represent function calls, but represent the events occurred in an application's life cycle. The OS respond to those events and determine the transition for applications.

Similarities and Differences. In the Android and Windows power models, the applications have both active and inactive states. The transitions between these two states are done via system API calls and managed by the developers. In Android, the power management is done by invoking *acquire* and *release* wakelocks. In Windows, the transition is enabled by system APIs that set the power state for either applications itself or for the device. In the latter case, the change can affect other running applications. Both these models are different from the iOS power model, in which the application transition through states mainly via events triggered during application execution. Developers have minimum controls for the state transition in that they only can manage the time used for state transition but cannot specify which states to transition to.

To further illustrate the differences and similarities of the three power models, in the following, we provide a coding example to show how Android, Windows implements *dim the backlight* respectively.

```
//Get powermanager, create new wakelock with
//low power setting, acquire wakelock
PowerManager pm = (PowerManager) getSystemService(
      Context.POWER_SERVICE);
mWakeLock = pm.newWakeLock(PowerManager.
      SCREEN_DIM_WAKE_LOCK);
mWakeLock.acquire();
```

Listing 1: Android Implementation for Lower Power State

```
\\D1 indicates low power
SetDevicePower(deviceName, POWER_NAME,
      DevicePowerState.D1);
```

Listing 2: Windows Implementation for Lower Power State

```
    //get current screen, set background color
    //to black, set alpha to 0.5,
    //apply view
    UIView *dimView = [[UIView alloc] initWithFrame:
        CGRectMake(0, 0, 320, 460)];
    dimView.backgroundColor = [UIColor blackColor];
    dimView.alpha = 0.5f;
    [self.view addSubview:dimView];
```

Listing 3: iOS Implementation for Lower Power State

The above code shows that while both Windows and Android achieve the *dim screen* effect by invoking certain system calls with appropriate parameters, the iOS does not have a similar mechanism that can be directly accessed by the developers. To achieve the same *dim lighting* effect, the programmer has to resort to the graphics library. In fact, in iOS, there are system functions that can achieve this effect, but they are private system functions that are not meant to be used. More function calls and setting configurations must be performed before these system functions can be used.

III. Comparing and Analyzing Energy Bugs

A. Methodology: Collecting Energy Bugs

Our next step is to investigate whether the similarities and differences discovered for power models will be reflected in the energy bugs reported at each platform. To do so, we manually select a set of energy bugs from Android, iOS and Windows applications. Our general methodology is as follows. We first select applications that commonly exist in three platforms. For each energy bug reported in Android, we search whether there are similar energy bugs reported in Windows and iOS for the same application. We believe that if energy bugs share a root cause across platforms, it is more likely to be discovered if we analyze the energy bugs from the same application running in different platforms rather than comparing energy bugs across different applications.

Data Source. As an initial study, we selected 6 real-life energy bugs for the Android platform from Facebook, Telephony, Android Exchange, Google Backup, Google Maps and Agenda Widget respectively [2]. To search for energy bugs in iOS, we looked into the open bug repository OpenRadar [9] and the discussion forum at the Apple site [10]. Typically, the developers that post bug information to the main Apple site will also repost it in the OpenRadar repository. For Windows, we did not find any bug repositories and forums. Our effort has been then searching for the third-party forum, called Windows Phone Central [11]. For both the OpenRadar repository and Windows Phone Central forum, we used the key words *battery drain*, *power drain* and *energy drain* along with the name of the application to discover relevant bug posts.

Data Validity. To increase the data validity and ensure that bug posts reflect the real energy problems a user encountered, we only consider the bugs that satisfy the following four conditions: 1) a user stated that the problem would cause the phone's battery to die out quickly, 2) there exists some description on what may cause the battery drain, 3) the cause is related to applications rather than hardware or OS problems, and 4) repeated posts about the same bug have been found in the forums.

Data Analysis Method. To compare Windows and iOS bugs with Android bugs and determine whether the energy bugs for three platforms share a root cause, we analyze the bug posts in three steps. First, we mapped the nouns of the bug descriptions from one bug to another and checked if their corresponding verbs are synonyms of each other. For example, an Android bug post described with *an application that keeps running in the background* is considered a match with an iOS bug post that says *the application runs continuously in the background*, as the two both have the key words *background* and similar verbs *keeps running* and *runs continuously*. We select such similar bug posts from different platforms into the data set. The second comparison of bug posts is performed at the API level. We extract the API calls and parameters related to wakelock from the Android bug posts, and determine if in the iOS and Windows bug posts, there are discussions related to misuse of API calls similar to wakelock. In the third step, we read the bug posts and identify the root causes for the energy bugs, especially for understanding how the power models are correlated with the behaviors of energy bugs for each platform. To address the problem of ambiguity, the conclusions of bug analysis were crosschecked by all three members of the team, and must be agreed by everyone.

B. Results: Energy Bugs and their Root Causes

Through our data collection, we identify a total of 15 energy bugs from three platforms, 6 from Android, 6 from iOS and 3 for Windows, shown in Table I. In this section, we present our results on analyzing these energy bugs, including the root causes of the energy bugs as well as the coding patterns related to the bugs.

Android. According to Pathak et al. [2], there are two main types of energy bugs occurring in Android applications, namely *no sleep bugs* and *loop bugs*. A no sleep bug occurs when an app wakes the CPU up but never puts it back to sleep due to mistakes in manipulating the wakelock APIs. The bug would cause a phone to excessively consume energy, even when no activities are performed. The no sleep bugs can be further classified to 1) no sleep code path, where the developer has acquired the wakelock, but has forgotten to release it along some program path, 2) no sleep race condition, where the acquire and release of wakelock follows an incorrect order due to the the race condition of threads, and 3) no sleep dilation, where the release of the wakelock is delayed, especially in a long running application. Loop bugs occur when a thread is waiting for an event to occur in order to continue, and a variable is used to test for a condition. The thread continues to poll the variable until the value changes, thereby unnecessarily consuming CPU cycles while not accomplishing any work. In Table II, we show the classification of the energy bugs for the three platforms. The results of the Android bugs in the second column are directly taken from Pathak et al.'s paper [2], used for comparing the energy bugs found in the Windows and iOS platforms.

Windows. In Section II, we show that the power models for Android and Windows have similarities. Thus, we expect to see similar behaviors and causes for energy bugs occurring in the two platforms. Our results support this hypothesis. From the initial data, we also found a loop bug on the Windows platform. See the Facebook bug under *Windows* in Table II. In addition, we found a type of energy bugs specific to Windows platform, namely *unnecessary power quota*. See the Mail Service bug under *Windows* in Table II. According to the Windows power model shown in Figure 2, an application can request for the minimum power requirement to execute.

TABLE I: Bug Comparison Table

App	Android	Windows	iOS
Facebook	Contains a facebook service that acquires the wakelock keeps the app running in the background even when not required [2].	The app spontaneously creates a core service loop, making the app continuously use up CPU causing battery drain [11].	A sync tool, that keeps on running continuously in the background leading to similar kind of power drain [10]
Cloud Backup	Cloud services acquire wakelock leads to power drain as data is synced. Worsens with network condition [2].	Windows phone has no app that contains an automatic service that backs-up data to the cloud [11]	iCloud service pushes data onto the cloud even when the phone is sleeping drains power [10]
Mail Service	Race condition during eMail synchronization [2].	Each additional email account that gets synched will contribute to a bigger battery drain [11].	Forces to sync with the email server at regular intervals [10]
Google Maps	Holds wakelock for hours, along with radio communication of GPS and network [2].	N/A	Services runs in background causes battery drain [10]
Calendar	Acquires wakelock to sync with web client [2].	N/A	For calendar entries that fail to sync, app keeps on resynching battery drain [10]
Telephony	Android telephony does not release the partial wakelock right away if there is an error in sending RIL(Radio Interface Layer) request [2].	N/A	Upon combining cellular and Wi-Fi capabilities together. Even when on Wi-Fi, it still exchanges data over cellular draining power [10]

TABLE II: Bug Classification Mapping

App	Android	Windows	iOS
Facebook	no sleep - Code Path	Loop Bug	Prolong State
Cloud Backup	no sleep - Dilation	(unknown)	Prolong State
Mail Service	no sleep - Race Condition	Unnecessary Power Quota	Prolong State
Google Maps	no sleep - Dilation	-	Prolong State
Calendar	no sleep - Code Path	-	Prolong State
Telephony	no sleep - Code Path	-	Prolong State

An unnecessary power quota bug occurs when an application demands higher power mode to perform long running task, whereas the application could function in unattended mode or lower power state, thus draining the battery rapidly than normal. There is another category of Windows energy bugs [6], which is similar to no sleep bugs in the Android platform. By default, the Windows mobile OS changes power state to sleep mode after a brief period of inactivity. An energy bug can occur when applications prevent the device from sleeping by periodically resetting the timer. It is our future work to find more energy bugs on the Windows platform to validate these types.

iOS. The iOS system revolves around the concept that an app can only be in one of four states. There are no APIs that control how these states are changed. The developers are responsible to capture the events and notify the system that an app is about to enter or leave a state. It should be noted that in iOS, when an app is in the process of transitioning states, it is still considered in the current but not next state. For example, during the transition from the *foreground active* state to the *foreground inactive* state, the app is still at the *foreground active* state. Energy bugs can occur if the developers incorrectly prolong the transition and keep the apps in foreground active and background active states, which we call *prolong state*. At the code level, this might be caused by developers incorrectly set the timer in the beginBackgroundTaskWithExpirationHandler API, for instance. Our analysis shows that all the iOS energy bugs under study belong to this type, show in the column under *iOS* in Table II.

From the bugs we studied, we also identified some of the coding patterns in applications that can cause battery drain. In Listings 3 and 4, we compare the implementation of the loop bug in Android and Windows platforms. Listing 3 shows a method written on the Android platform where a loop keeps checking the value of the wakelock. If the value is equal to a value specified by the developer, the method can complete. Listing 4 shows a method written on the Windows platform where there's a loop that keeps checking the power value of the device. If the power value of the device is equal to a power value specified by the developer, then the method can complete. Although the code is different in the two examples as the two platforms use different languages, we can identify their similarity at the design level. Providing these coding patterns, we potentially enable static analyzers to detect these energy bugs.

```
public void longRunningMethod( Object o) {

  while( someVariable != certainWakeLockValue ) {
    someVariable = PowerManager.WakeLock
  }
  doSomething();
}
```

Listing 4: Android Implementation of Loop Bug

```
public void longRunningMethod( Object o) {

  while( someVariable != somePowerValue ) {
    GetDevicePower(LPWSTRDeviceName, POWER_NAME,
        someVariable );
  }
  doSomething();
}
```

Listing 5: Windows Implementation of Loop Bug

IV. Related Work

Pathak et al. performed the first study to introduce the community about classifying energy bugs in smartphone apps [1]. The research focuses on surveys and investigations of repositories and multiple forums on bugs that contain characteristics of unusual battery drain. It provides a taxonomy of Android energy bugs caused by hardware, OS and app configurations. Their goal was about defining the existence of energy bugs but not comparing energy bugs across different platforms.

Pathak et al. also performed the first study on characterizing no sleep energy bugs for the Android platform [2]. They give an in-depth analysis of how bugs are made in the Android platform and identified the root cause as power encumbered programming leading to mishandling of the power control APIs by programmers. However, this study only examines the Android platform and doesnt even mention other smart phone platforms. Our research has taken the method of collecting bugs and examining the API in the same fashion as this paper, but applying it to the Windows and iOS platforms.

Pathak et. al. investigated and answered the question to how energy is spent in an application [3]. They implemented eProf - an energy consumption profiling tool that accounts for how energy is spent among program entities: threads, subroutines, processes, and system calls on Windows and Android platforms, and proposed a new presentation of energy accounting, which helps app developers to quickly understand and optimize the I/O energy drain of their apps.

We investigated our research questions in two steps. To answer our first research question given in Section I, we studied Windows and iOS APIs to find out if both platforms expose power control APIs as the ones in the Android framework. To answer our second research question, we focused on 6 specifically different apps and analyzed their energy bugs occurred in the three different platforms.

V. Discussion

Although in this study, we chose popular applications, such as Facebook and GoogleMaps, we still have challenges to find a number of energy bugs for the Windows platform. First, any data about bugs on the Windows phone would have to be found on the third party websites. Also, a Nielson [12] study shows that the majority of Smartphones sold in the world during the Q2 period of 2012 have been Android and iOS based phones, with each system selling in over 30% of all smartphones sold. Windows, however, falls below 10% of all smartphones sold. We thus have a smaller sample size. For iOS, we find that the majority of the bugs reported in the OpenRadar repository were OS specific. As a result, the discussion forums on the apple site became our primary source of bug collection.

VI. Conclusions and Future Work

This paper presents an initial study on comparing power models and energy bugs for the Android, Windows and iOS platforms. We found that for all of the three platforms, the developers can make mistakes during coding and introduce energy bugs, as these smartphone platforms all provide system APIs for the developers to access the power management. However, on the Windows and Android platforms, the developers have more controls on the applications' power usage, while the iOS platform only provides minimum flexibility for the developers. Furthermore, the Windows power model shows a similarity to Android's for changing power states for applications. As a result, both Windows and Android platforms have no sleep bugs and loop bugs that can cause battery drain. The iOS platform, however, has a different power model, where the developers cannot directly change power states for applications, but only can delay the transition, which causes a different type of energy bugs.

This work provides the insight for the tool builders. The implication is that the techniques for managing energy bugs developed for the Android platform may be directly applicable to the Windows platform, but we might need to develop new tools for detecting and mitigating energy bugs in iOS applications. Hopefully, a holistic understanding across several platforms will open further field of study to mitigate the problems, e.g., building cross-platform language capabilities to avoid energy bugs at the software design level. We have planned several directions for the future work. We will spend more effort in collecting energy bugs in Windows and iOS. We consider implementing the bug patterns discovered and measure the severity of these energy bugs. We also may extend the scope of our research to other smartphone platforms.

References

[1] 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*, ser. HotNets-X. New York, NY, USA: ACM, 2011, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2070562.2070567

[2] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307661

[3] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168841

[4] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying android apps for the absence of no-sleep energy bugs," in *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, ser. HotPower'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387869.2387872

[5] M. Inc., "Msdn: Power management functions," http://msdn.microsoft.com/en-us/library/aa909892.aspx.

[6] J. I. Johnson, "Windows mobile power management," http://www.codeproject.com/Articles/28886/Windows-Mobile-Power-Management.

[7] Apple Inc., "Ios developer library," https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html.

[8] Google, "Android power manager api," http://developer.android.com/reference/android/os/PowerManager.html.

[9] OpenRadar, "Openradar bug repository," http://openradar.appspot.com/.

[10] D. Apple, "Apple discussion forums," https://discussions.apple.com/.

[11] W. D. Forums, "Windows discussion forums," http://forums.wpcentral.com/.

[12] Nielson, "Two thirds of new mobile buyers now opting for smartphones," http://blog.nielsen.com/nielsenwireonline_mobile/two-thirds-of-new-mobile-buyers-now-opting-for-smartphones/.