IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 10, 2018, accepted September 21, 2018, date of publication October 22, 2018,
date of current version November 30, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2877395

Characterization and Detection of Tail Energy Bugs in Smartphones

ABDUL MUQTADIR ABBASI[1], MUSTAFA AL-TEKREETI[1], KSHIRASAGAR NAIK[1], AMIYA NAYAK[2], PRADEEP SRIVASTAVA[3], AND MARZIA ZAMAN[3]

[1]Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L3G1, Canada
[2]School of Information Technology and Engineering (SITE), University of Ottawa, Ottawa, ON K1N 6N5, Canada
[3]Technologie Sanstream, Montreal, QC J8Y 2V5, Canada

Corresponding author: Abdul Muqtadir Abbasi (amabbasi@uwaterloo.ca)

This study was supported by the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT Smartphones are the most ubiquitous and popular hand-held devices because of their rich set of features and wide variety of services that require frequent battery recharging. In smartphones, most energy issues are due to energy bugs (ebugs). These ebugs are said to be exist when smartphone software applications (apps) consume more power than expected while executing, or they continue to consume energy even after these apps are closed or terminated. In this research, we introduce the concept of application tail energy bugs (ATEBs) and provide an operational definition for it. Then, we discuss about the main potential causes of ATEBs and the user actions that can trigger them. To provide a proof of concept, we conduct experiments using real Android apps. To identify all the scenarios that can cause ATEBs, we develop a testing app and perform 32 experiments. The main goal of the experiments is to check the behavior of app components, such as activities and services in the presence/absence of four different types of wakelocks. Then, we discuss the relationship between software changes and energy consumption by tracing wakelocks that keep the device awake and services that might be engaging the CPU. The power consumption of the app is measured using the monsoon power meter. Because power meters are not often available to software developers, we design a tool to detect ATEBs. This tool utilizes Android debug bridge (adb) commands to extract system-related information. The tool effectiveness is evaluated using five Android apps.

INDEX TERMS Energy bug, software testing, energy efficiency.

I. INTRODUCTION

Mobile devices are pervasive due to their rich features and services. However, they are constrained by their battery life. Unfortunately, the development in battery technology is not on par with the developments in software and hardware systems [1]. Therefore, a paramount attention from both academia and industry has been given for developing more power efficient techniques and algorithms in both hardware and software. Due to high market competition, app developers rarely have sufficient time to carefully optimize their app's energy consumption [2]. Thus, many apps suffer from energy bugs. In the presence of an energy bug, the app may not fail/stop, but it may only cause a higher energy consumption, which makes energy bug very difficult to detect [3]. In literature, there is a lack of useful information which can guide developers how to detect the presence of energy bugs in smartphones as shown in the next section.

In this work, we consider Android based smartphones. Android is an open source platform that has dominated the market very quickly. For power optimization purposes, Android apps do not provide an explicit "Exit" or "Quit" button. It has been reported that frequently loading apps into and out of the memory has a negative impact on the battery life. When the app is stopped, Android OS tries to keep it in the memory as long as possible. When the system is out of memory, the OS starts to kill stopped apps to reclaim space. Recently used apps start more quickly if they are still in the memory. Instead of an explicit exit button, Android phones provide multiple ways to close a running app, such as pressing the home/back button, using the swipe-out gesture, or using the force-stop option from the settings. Having many ways to close the app has created confusion among users about the best way to close the app. Moreover, It has been noticed that different apps have differently implemented app closing options. This inconsistency in app implementation has

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

negatively affected the battery life and thereby the experience of the end user.

In addition, because smartphones are resource limited devices, mishandling resources has often caused power consumption inefficiencies. In this work, we are interested in the type of energy bugs that sustain after closing the app. We call this type of energy bugs as the app tail energy bug (ATEB). As the name suggests, an ATEB is energy wastage due to some unwanted processing after stopping or closing the app. This energy loss indicates that the app does not allow the phone to sleep, resulting in prolonged battery drain.

In this research, we discuss the potential causes of ATEB. Then, we show how certain user actions in closing the app can trigger the ATEB. We conduct experiments using real Android apps from Google play store. We also explore the interaction between programmer's mistakes and user's actions. The power consumption of the app is measured using the Monsoon power meter. Then, a tool is designed to detect ATEBs without the need to use the power meter. This tool utilizes Android debug bridge (adb) commands to extract system related information. The tool effectiveness is validated through power consumption measurements using the power meter. In summary, we make the following three contributions:

- We provide a formal definition of the app tail energy bug (ATEB);
- We identify the potential root causes of ATEBs and the user actions that trigger them; and
- We develop a Java based tool to detect the presence of app tail energy bugs in Android smartphones.

The rest of the paper is organized as follows. In Section II, we present the research literature related to app-tail energy bugs in smartphones. In Section III, we present the operational definition of ATEB. In Section IV, we validate the definition of app tail energy bugs using real apps. In Section V, we apply use cases to simulate more ATEBs. In Section VI, we present ATEBs tool design and evaluate its effectiveness. Finally in Sections VII and VIII, we discuss the results and conclude the work.

II. RELATED WORK

In order to make smartphone applications more energy efficient, researchers have worked mostly from three perspectives. First, various definitions have been proposed to characterize energy bugs. Second, different frameworks are proposed to detect energy bugs. Finally, to reduce total cost of testing for energy bugs and automate the process, different tools are designed. In this section, we review the published works that fall into these three prospectives.

The available definitions of energy bugs are very broad, emphasizing only the causes using non measurable descriptive words. Researchers have discussed different types of energy bugs that manifest after closing the app, such as resource leaks [4], wakelock bugs [5], vacuous background services bugs [6], and immortality bugs [3]. Even though guidelines to locate energy bugs are provided, the necessary line of actions is not available. Pathak *et al.* [3] presented a taxonomy of energy bugs in smartphones and discussed the reasons for those energy bugs. In our previous work, an operational definition for energy bugs is provided [7]. In this work, we utilize this definition and propose a framework to detect a special type of ebugs which is app tail energy bugs.

In literature, a number of different frameworks have been developed. Pathak *et al.* [3] proposed guidelines for developing a systematic diagnosis framework to detect energy bugs in smartphones. They recommended to classify energy bugs on the basis of symptoms and then identify the faulty software component, but they did not provide a workable procedure that developer can apply. Banerjee *et al.* [6] proposed a framework to systematically generate test inputs that help to capture energy bugs. Each test input is basically a sequence of user interactions (touches or taps on a smartphone screen) that may trigger an energy bug in the application. They used a customized version of a third-party tool to generate control flow graphs for event trace generation. The tool does not cover all possible GUI states in the application. The drawback is the lack of selection criteria for the suspected user inputs which can stress energy bugs in smartphones, whereas, in our work, we identify the user actions that need to be covered in order to trigger an ATEB.

In literature, the main focus is to design tools to detect wakelock bugs. Pathak *et al.* [5] used dataflow analysis technique to detect energy bugs caused by wakelock leakage. Later, researchers proposed different static and dynamic analysis techniques for the same purpose. For example, Vekris *et al* [8]. proposed a static analysis technique for verifying the absence of wakelock leakage in an Android app. Static analysis requires the availability of the app code, which limits the applicability of the tool. In [9], a Wakescope scheme is designed to detect and notify of a misuse of wakelock handling. In [10], EnergyPatch is developed that uses a combination of static and dynamic analysis techniques to detect, validate, and repair energy bugs in Android apps. Our tool is more close to WLCleaner [11]. WLCleaner uses adb commands to fetch wakelocks status from the kernel. It also checks CPU utilization after the screen is off. If the CPU is not running, the application is in the sleep mode and any active wakelock must be released, whereas in our tool, we compare the system states before starting and after closing the app to detect unreleased wakelocks. Moreover, WLCleaner is an application that can be installed in smartphones the run Android older than Android 4.4 KitKat. Starting from Android 4.4 KitKat, Google has tighten the OS security by removing in-device adb support. Therefore, we develop a Java based desktop tool by utilizing embedded adb commands.

III. APPLICATION TAIL ENERGY BUGS

In this section, we propose an operational definition for the app tail energy bug. An app is considered to have caused an ATEB in a smartphone when the mean of power consumption (P_f) after closing the app is greater than

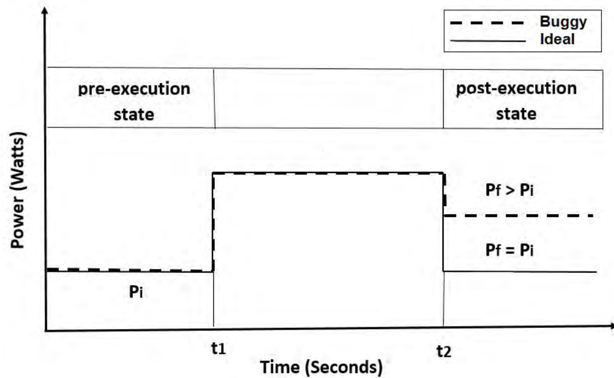

FIGURE 1. Power trace for a hypothetical application.

the mean of power consumption (Pi) before opening the app.

The concept of power loss or leakage due to a hidden or unseen defect is very broad and it is applicable in different fields [12]. Figure 1 shows a power consumption trace of a hypothetical app. We define two states with respect to the power consumption of the app: *pre-execution state* and *post-execution state*. For both states, the app power print is characterized by the mean rate of energy consumption which represents the power in watts (P_i and P_f). According to the definition, a smartphone is in an ideal working condition without the presence of an application tail energy bug if the power consumption of pre-execution state and the post-execution state are the same. That is:

$$P_f = P_i \tag{1}$$

Now, if there is an application tail energy bug in the smartphone, the desired output will not be achieved and the system may be in a high power consuming state. In this state, the smartphone consumes on average more power than it was before the execution of the app. It will lead smartphone's battery to drain. Thus:

$$P_f > P_i \tag{2}$$

IV. IDENTIFICATION OF APPLICATION TAIL ENERGY BUGS

In this section, we first demonstrate the test bench setup used to validate the definition of ATEBs. Then, we discuss how programmer actions can lead to ATEBs. Lastly, we identify a set of user actions that can trigger ATEBs. We do that by means of experiments and real mobile apps from Google store.

A. TEST BENCH

The test bench setup for the validation of our proposed definition for ATEBs is shown in Figure 2. The smartphone is energized from the power meter to provide a constant voltage source. A laptop computer is used to continuously

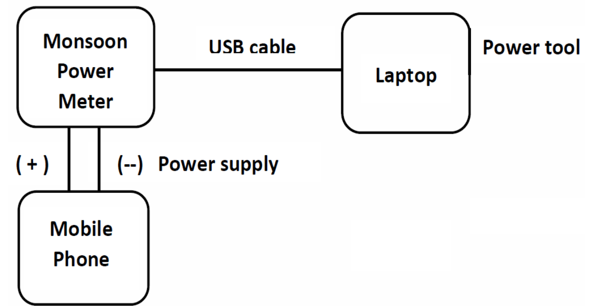

FIGURE 2. The test bench with Power tool.

monitor the power supply and collect the measurement data for the test duration using the Power tool app. To bypass the smartphone's battery, a modified battery connection is made to power the smartphone externally. We set up this connection to disconnect the battery's power interface, but keep the data interface connected to the smartphone. To avoid the power saving mode and for accurate results, the smartphone's battery is kept fully charged [13].

B. POTENTIAL CAUSES OF ATEBs

An ATEB can be caused by various programming faults. Programmer errors in coding create a flaw in the app, which could activate an ATEB. It is reported in more than one place that wakelocks and Services mishandling are the main source of bugs that can lead to ATEBs in Android devices [14]. In this work, we further investigate the interaction of wakelocks and Services with user actions and we identify test scenarios that may lead to ATEBs. We focus on programming errors that affect the life cycle of the two main components of Android apps: Activities and Services.

C. USER ACTIONS

Testing for ebugs is challenging due to the difficulty of verifying the output of test cases (test oracles [15]). Therefore, in this work, we focus on the type of ebugs that sustain after closing the app, utilizing the fact that the mean difference in power consumption after closing the app should return back to the mean difference in power consumption before launching the app if there is no ebug. In Android devices, there are five actions that can close the app and thereby help in detecting ATEBs. They are:

- pressing the home button,
- pressing the back button,
- using the swipe-out gesture,
- using the force-stop option from the settings, and
- using the exit button if available.

The home and back buttons on the virtual task bar may seem to close apps quickly, but in fact, they actually just minimize them. The minimized app is still technically running (usually in some sort of "paused" state), and therefore, it can be reloaded quickly. It is recommended to avoid

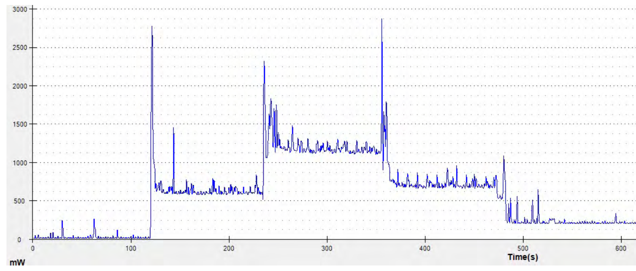

FIGURE 3. **A power trace of the Radio FM app with a wakelock ATEB.**

using the home and back buttons when the goal is to end processes and conserve device's resources. The force-stop option is handled by the OS. It closes the app completely including Activities and Services and it releases all types of wakelocks.

D. VALIDATION

In this section, we show how ATEBs can be triggered if the app is ended using one of the actions in Subsection C. By conducting ten experiments, we find that certain user actions can help in triggering certain types of ATEBs. In the remaining part of this section, we explain each of these scenarios by means of real power measurements. We quantify the difference in power consumption (P_L) between the buggy behavior and the ideal behavior using the following relationship:

$$P_L = ((P_2 - P_1)/P_1) * 100\% \tag{3}$$

where P_2 is the power consumption of the buggy behavior and P_1 is the power consumption of the ideal behavior. We use for this purpose two Android apps. To check for wakelock related problems, we have selected Radio FM app of version 5.6. It is an Internet based radio service. To check for Service related problems, we have selected the Aripuca GPS tracker of version 1.3.4. It is a free and open source GPS tracking app for Android.

1) SCENARIO 1 (ATEB DUE TO WAKELOCKS MISHANDLING)

We have observed an ATEB with Radio FM app due to wakelocks. When the user tries to close the app using the swipe-out gesture, audio service is not stopped completely due to an active partial wakelock. Mostly, users like to play with multiple apps while using a music or radio app. Later on, they forget to close it completely assuming that when they close the radio or music app the audio service would have been stopped as well. In reality, audio services never stop by itself and drain the battery power in few hours.

Figure 3 shows the actual power consumption behavior of the Radio FM app and Figure 4 shows the averaged power consumption behavior for the same app. We will only show the averaged power consumption in the subsequent experiments. The test scenario is the following. At the beginning, the smartphone is in the sleep mode. At T1, we press the power button to start the phone. At T2, we launch the app. At T3, we select a radio station and the radio service starts. To keep the audio services running continuously, a partial wakelock is acquired. At T4, we stop the app using two ways. For the buggy behavior, we close the app by pressing the home button and then the swipe-out gesture. For the ideal behavior, we end the app using the force-stop option. As shown in Figure 4, there is a difference in power consumption due to that the audio service is still running in the buggy behavior. Using Equation (3), the mean difference in power consumption P_L is 10.5%. In the ideal behavior, the device goes into the sleep mode at T5 when the app is completely closed using the force-stop option.

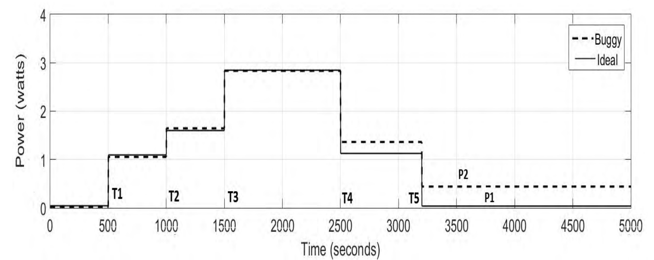

FIGURE 4. **The smoothed power traces of the Radio FM app.**

2) SCENARIO 2 (ATEB DUE TO SERVICES MISHANDLING)

We have observed an ATEB with Aripuca GPS tracker due to mishandling of Services. When the user tries to close the app using swipe-out gesture, location services are not stopped. An active listener is keeping the location services running in the background. We conduct an experiment to check the power consumption behavior of the app especially once the app is not visible. In order to check any deviation from the ideal behavior, we have tested two versions of the same app (1.3.4).

Figure 5 shows the power consumption behavior of Aripuca GPS tracker app. The test scenario is the following. At the beginning, the smartphone is in the sleep mode. At T1, we press the power button to start the phone. At T2, we launch the app and select a waypoint "Niagra fall" from the way-

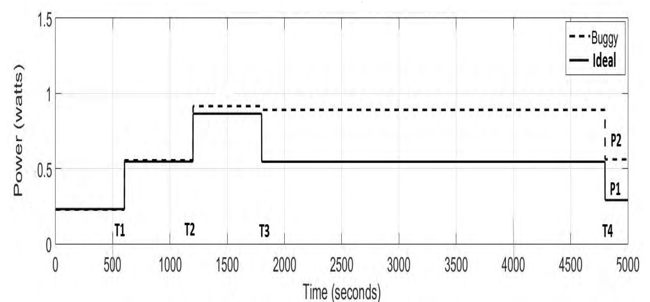

FIGURE 5. **Power traces of Aripuca GPS tracker.**

points screen. At T3, we stop the app using two ways. For the buggy behavior, we close the app by pressing the home button and then the swipe-out gesture. For the ideal behavior, we end the app using the force-stop option. As shown in Figure 5, there is a difference in power consumption due to that the location services are still running in the buggy behavior. The mean difference in power consumption P_L is about 53%. In the ideal behavior, at T4, the device goes into the sleep mode when the app is completely closed using the force-stop option from the settings.

V. USE CASES TO SIMULATE ATEBs

To further explore whether there are more scenarios that can lead to ATEBs, we design a testing Android app and conduct 32 experiments. Figures 6 and 7 show the GUIs and the control flow of the testing app, respectively. The app has two Activities and one Service. Each Activity corresponds to a screen. In one screen, we can acquire and release four types of wakelocks. Table 1 shows the four supported types of wakelocks and the associated resources when the wakelock is active. Thus, we can study the interaction between the Activity component and wakelocks. In the second screen, we launch a Service to download a file. We can also acquire and release wakelocks to complete the download task. Thus,

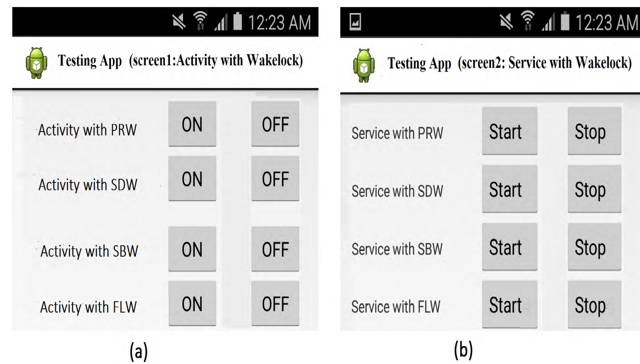

FIGURE 6. The two GUIs of the testing app, where PRW, SDW, SBW, and FLW are abbreviations for partial wakelock, screen dim wakelock, screen bright wakelock and full wakelock, respectively.

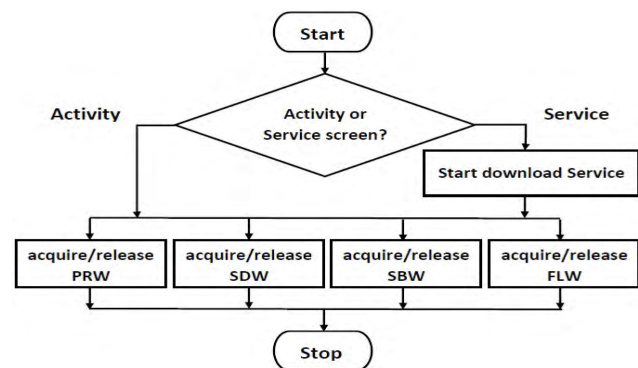

FIGURE 7. The flowchart of the testing app, where PRW, SDW, SBW, and FLW are abbreviations for partial wakelock, screen dim wakelock, screen bright wakelock and full wakelock, respectively.

TABLE 1. Wakelock types.

Wakelock type	CPU	Screen	Keyboard
Partial wakelock	Yes	No	No
Screen dim wakelock	Yes	dim	No
Screen bright wakelock	Yes	bright	No
Full wakelock	Yes	bright	Yes

we can investigate the interaction between the Service component and different types of wakelocks. Google has provided PowerManager API to control smartphone sleeping behavior using wakelocks. Each wakelock type has a different impact on the system power consumption. The partial wakelock is the most critical one because its effect is not visible to the user. Moreover, in the partial wakelock, the user cannot force the system to go to the sleeping mode by pressing the power button as in the other types of wakelocks. Using 32 test cases, we investigate the impact of mishandling of four types of wakelocks in an Activity or Service component on the power consumption behavior. The app is ended by pressing the home button, back button, swipe-out gesture, or force-stop option. There are two test cases with Activity and three with Service that exhibit ATEBs. A summary of the experiments is shown in Table 2 and Table 3. The third column represents the action used to end the app. For some experiments, such as experiments three and four in Table 2 and Table 3, the mean difference in power consumption (P_L) is significantly higher than other experiments due to mainly the energy cost of keeping the screen bright.

TABLE 2. Summary of results for the activity component with Wakelocks.

Exp No	*Wakelock type*	*User action*	*Power loss*
1	Partial	home button	4.7%
2	Screen dim	home button	26.7%
3	Screen bright	home button	838.1%
4	Full	home button	955.7%
5	Partial	back button	5.0%
6	Screen dim	back button	26.7%
7	Screen bright	back button	838.1%
8	Full	back button	955.7%
9	Partial	swipe-out gesture	0%
10	Screen dim	swipe-out gesture	0%
11	Screen bright	swipe-out gesture	0%
12	Full	swipe-out gesture	0%
13	Partial	force-stop	0%
14	Screen dim	force-stop	0%
15	Screen bright	force-stop	0%
16	Full	force-stop	0%

In the remaining part of this section, we explain each of these scenarios by means of real power measurements. Activities do not show ATEBs in the absence of wakelocks. However, in the case of Services, there is a possibility of an ATEB if the app is designed to keep the service running in the background while the app user does not know this requirement. This kind of mismatch between user expectation

TABLE 3. Summary of results for the service component with Wakelocks.

Exp No	*Wakelock type*	*User action*	*Power loss*
1	Partial	home button	5.0%
2	Screen dim	home button	26.7%
3	Screen bright	home button	838.1%
4	Full	home button	955.7%
5	Partial	back button	5.1%
6	Screen dim	back button	26.7%
7	Screen bright	back button	838.1%
8	Full	back button	955.7%
9	Partial	swipe-out gesture	5.0%
10	Screen dim	swipe-out gesture	26.7%
11	Screen bright	swipe-out gesture	838.1%
12	Full	swipe-out gesture	955.7%
13	Partial	force-stop	0%
14	Screen dim	force-stop	0%
15	Screen bright	force-stop	0%
16	Full	force-stop	0%

and design specifications might lead to energy wastage which can be considered as ATEBs.

3) SCENARIO 1

In this scenario, we investigate the impact of a mishandled partial wakelock in an Activity component on the power consumption behavior. The app is ended using the home button. The test scenario is the following. At the beginning, the smartphone is in the sleep mode. At T1, we press the power button to start the phone. At T2, we acquire the partial wakelock by pressing the "ON" button in the first screen. At T3, we stop the app using two ways. For the buggy behavior, we close the app by pressing home button. For the ideal behavior, we end the app using the force-stop option. As shown in Figure 8, there is a difference in power consumption due to an unreleased wakelock. In the buggy behavior, at T4, we observe the display is OFF which means the app is closed but it is not the case. A partial wakelock is still active and that is why the device does not go into the sleep mode. The mean difference in power consumption P_L is about 4.78%. In the ideal behavior, the device goes into the sleep mode at T4 when the app is closed at T3 using the force-stop button.

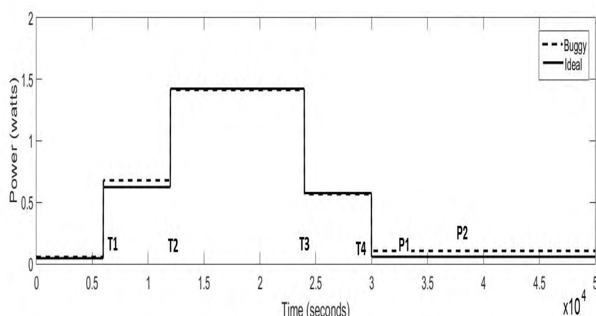

FIGURE 8. Power traces of the testing app (Scenario 1).

4) SCENARIO 2

In this scenario, we investigate the impact of a mishandled partial wakelock in a Service component on the power consumption behavior. The app is ended using the home button. The test scenario is the following. At the beginning, the smartphone is in the sleep mode. At T1, we press the power button to start the phone. At T2, we launch the app. At T3, we start the file download service by pressing the Start button in the second screen. At T4, we stop the app using two ways. For the buggy behavior, we close the app by pressing home button. For the ideal behavior, we end the app using the force-stop option. As shown in Figure 9, there is a difference in power consumption due to an unreleased wakelock. In the buggy behavior, at T5, we observe the display is OFF which means the app is closed completely, but it is not the case. A partial wakelock is keeping the device active and the Service to run continuously till T6 to finish its job. At T6, the partial wakelock is still active and this is why the device does not go into the sleep mode. The mean difference in power consumption P_L is about 5.09%. In the ideal behavior, the device goes into the sleep mode at T5 when app is completely closed at T4 using the force-stop option.

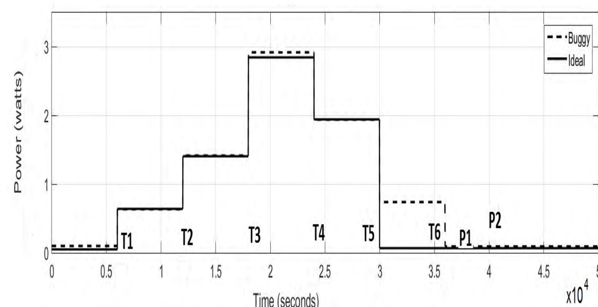

FIGURE 9. Power traces of the testing app (Scenario 2).

5) SCENARIO 3

In this scenario, we investigate the impact of a mishandled partial wakelock in a Service component on the power consumption behavior. The app is ended using the swipe-out gesture. The test scenario is the following. At the beginning, the smartphone is in the sleep mode. At T1, we press the power button to start the phone. At T2, we launch the app. At T3, we start the file download service by pressing the Start button in the second screen. At T4, we stop the app using two ways. For the buggy behavior, we close the app using the swipe-out gesture. For the ideal behavior, we end the app using the force-stop option. As shown in Figure 10, there is a difference in power consumption due to an unreleased wakelock. In the buggy behavior, at T4, the file download Service is relaunched automatically. At T5, we observe the display is OFF which means the app is closed completely, but it is not the case. A partial wakelock is keeping the device active and the Service to run continuously till T6 to finish

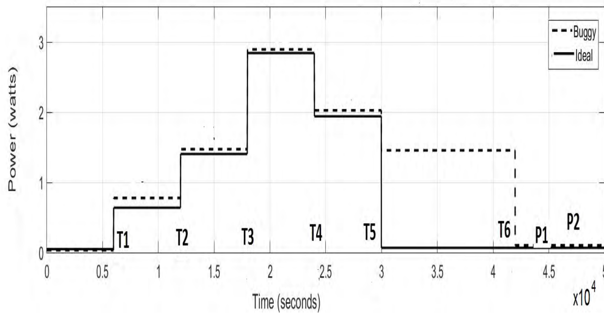

FIGURE 10. Power traces of the testing app (Scenario 3).

its job. At T6, the partial wakelock is still active and this is why the device does not go into the sleep mode. The mean difference in power consumption P_L is about 5.11%. In the ideal behavior, the device goes into sleep mode at T5 when the app is completely closed at T4 using the force-stop option.

In all the three scenarios, the power loss is relatively low because the acquired wakelock is partial. In case of other type of wakelocks, the mean percentage of power loss is much higher as shown in Table 2 and Table 3.

VI. APPLICATION TAIL ENERGY BUGS DETECTOR

In this section, we demonstrate a Java based tool called the ATEBs detector. This tool can be used to detect ATEBs without the need of using the power meter. It also does not require the availability of the app code. Then, we conduct experiments to show the effectiveness of the tool using power measurements as ground truth.

A. ATEBS DETECTOR DESIGN

To track the root causes of unexpected battery drain in smartphones and according to our definition of ATEBs, we need to capture the system state two times. The first time is before launching the app under test and the second time is after running the test scenario and closing the app. For device monitoring at runtime, Android platform has provided the Android Debug Bridge (adb) to initiate system level commands such as dumpsys, dumpstate and logcat. Due to security reasons, system information is no longer available to in-device apps. Starting from Android 4.4 KitKat, this information is only available through a USB port when the smartphone is set into the USB debugging mode. Because we do not want to break the security of the device, we design the ATEBs detector as a Java based desktop app.

Figure 11 shows the ATEBs detector block diagram. The tool consists of two main blocks: Android debug bridge (adb) and a parser. The input to the tool is a set of test scenarios. Each test scenario is designed to stress a certain app functionality. The important aspect is that each test scenario should start the app under test from scratch and close the app using one of the reported actions to end the app. The outcome of the tool is a report. This report contains a list of zero or more

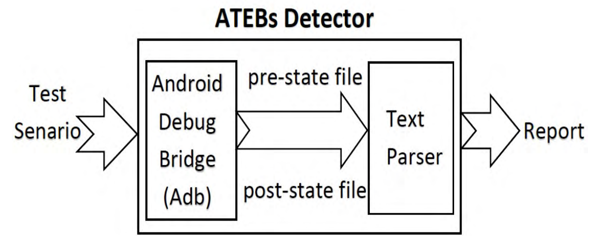

FIGURE 11. The ATEBs detector tool.

items. Each item is an indication of an ATEB. Therefore, the developer has to investigate each item of this list individually to detect whether it is triggered by the app under test or it is a possible false alarm.

Android debug bridge (adb) is the most powerful tool provided by Google to help developers for debugging and monitoring purposes. It provides access to the Unix shell that can be used to run a variety of commands in the device. It consists of three components. The first component is the client that runs on the development machine. The second component is the daemon (adbd) that runs commands on the device. The third component is the server that manages communication between the client and the daemon. Once the adb server receives system level kernel information from the adb daemon, ATEBs detector saves daemon's output as a log file in the desktop. We run adb commands twice: before running the app (pre-state file) and after closing the app (post-state file).

The second main component of ATEBs detector is the text parser. The main function of the parser is to compare pre-state and post-state files line by line and find any difference that could be an indication of something that still active and causing an ATEB. In our test bench, we run ATEBs detector on a laptop and the app under test on an Android smartphone that is connected to the laptop via a USB cable as shown in Figure 12. The power meter is used only to verify the output of the ATEB detector. Figures 13, 14, 15, 16, and 17 are examples of the output of the tool. The GUI of the tool is divided into two main sections: a button plate on the top and

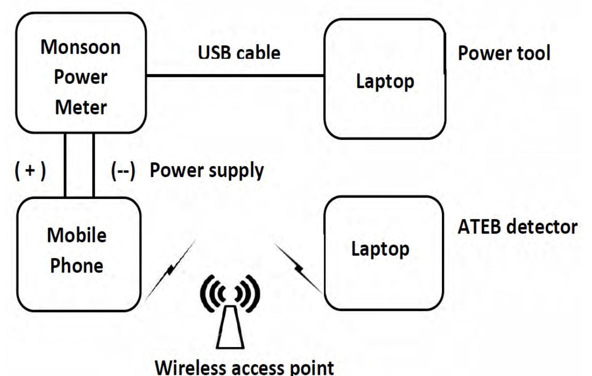

FIGURE 12. The test bench with the ATEB detector tool.

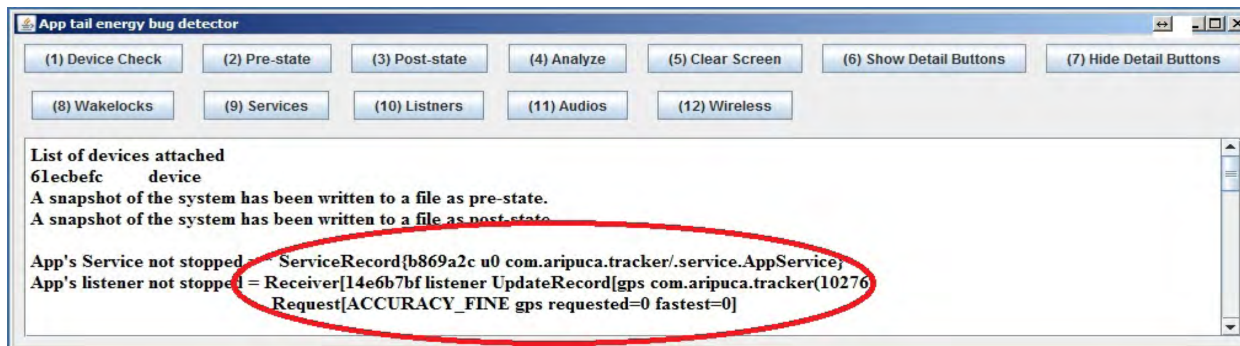

FIGURE 13. ATEB detetcor's report for Aripuca GPS tracker app.

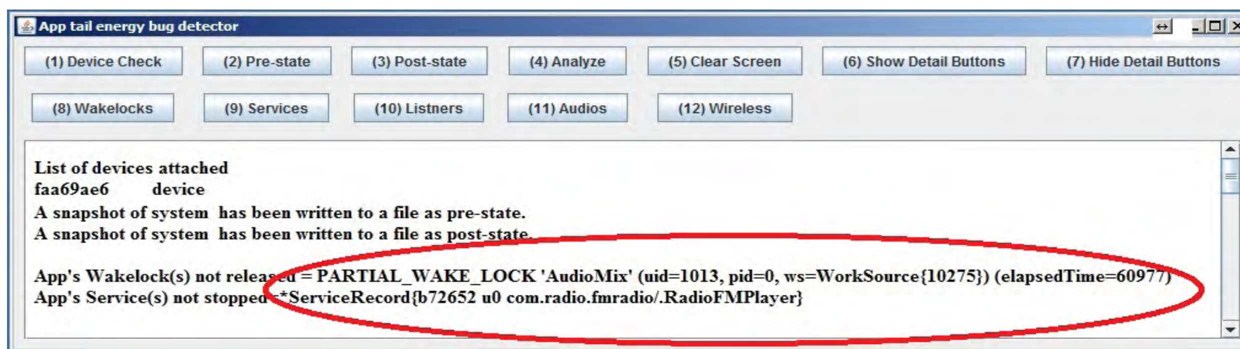

FIGURE 14. ATEB detetcor's report for Radio FM app.

FIGURE 15. ATEB detetcor's report for GPSLogger app.

a result pane in the bottom. To check for a specific type of ATEB, an individual button is provided for each type, such as wakelocks and Services.

B. ATEBs DETECTOR TOOL EVALUATION

To show the effectiveness of the tool, we have conducted experiments using five buggy Android apps. All the reported ATEBs are acknowledged by our ATEB detector. Figures 13 and 14 respectively show the output of ATEBs detector for Aripuca GPS tracker and Radio FM apps that are discussed earlier in Section IV. Using Monsoon power meter, we have detected the presence of an ATEB. However, we could not find out the root cause of the ATEBs. The main advantage of our tool is that it provides the verified in-depth system information to pinpoint the cause of the bug. Then, we apply the tool to test three more apps: GPSLogger app version 15, Kuwo app version 2.3.1.0, and Omnidriod app version 0.2.2. GPSLogger provides route information for the users and also tags photos with location coordinates. Kuwo is a Chinese music player app. It allows to search and play on-line music files. Omnidroid is an automated event/action manager that allows users to automate system functionality based on incoming application intents. It also provides a general plug-in framework that allows any compatible application events to trigger any other applications' actions. The results are shown in Figure 15, 16 and 17.

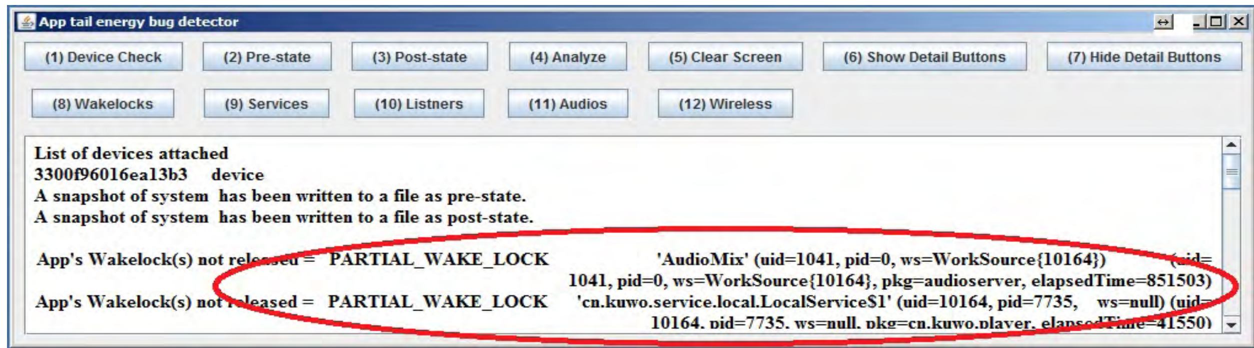

FIGURE 16. ATEB detetcor's report for Kuwo Music player app.

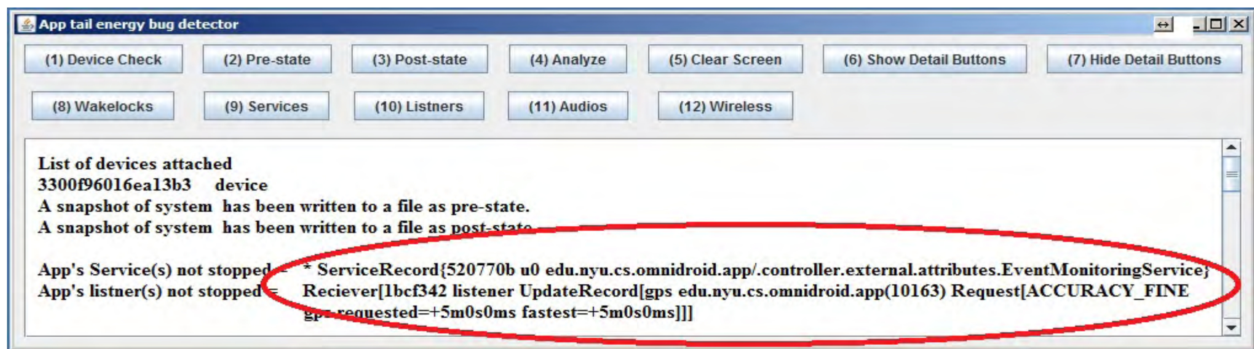

FIGURE 17. ATEB detetcor's report for Omnidriod app.

In Figures 13 and 14, a service is still running because a listener to location services is not released. It is a GPS service that consumes very significant amount of power. In Figures 17, a monitoring service is still running and consuming power. In Figure 14 and 16, a service is still running because a partial wakelock is not released. This information works as a guideline for the developer to rectify the problem.

VII. DISCUSSION

Mobile devices have become an integral part of people life. They offer a wide spectrum of services that range from entertainment to healthcare [16], [17]. The main challenge in detecting energy related software bugs is the absence of the required test oracles. To overcome this problem, we introduced the concept of app tail energy bug (ATEB). An app is considered to have an ATEB in a smartphone when the mean of power consumption (P_f) after closing the app is greater than the mean of power consumption (Pi) before opening the app. According to this definition, test scenarios are designed so that buggy power consumption is easily identified without the need to know in advance the expected power consumption. Because power meters are not often available to software developers, we designed and implemented a software based ATEBs detector. To detect ATEBs in Android smartphones, we used two different approaches: a hardware-based tool (Monsoon power meter) and an in-house built software-based tool (ATEBs detector). In the experiments, we ran both tools on the same laptop. However, all these solutions present their own advantages and disadvantages. Hardware-based tools are highly precise for power measurement, but they cannot provide the insight of system activities. They can confirm the existence of ATEBs, but they do not pin point the root causes. ATEBs detector requires kernel level system information to detect the root cause of ATEBs, but they cannot be used for the quantification of the power loss. Mostly, software-based tools require breaking device lock which is not permitted by service providers. Our tool do not require the smartphone to be unlocked.

VIII. CONCLUSION

In this research, we introduced the concept of application tail energy bugs ATEBs and provided an operational definition for it. Then, we discussed about the main potential causes of ATEBs and the user actions that can trigger them. To provide a proof of concept, we conducted experiments using real Android apps. To identify different causes of ATEBs, we developed a testing app and performed 32 experiments to check app components such as Activity and Service behavior in the presence or absence of four different types of wakelocks. Then, we discussed the relationship between software changes and energy consumption by tracing wakelocks that keep a device awake and Services that might be engaging the CPU silently. The power consumption of the app was measured using the Monsoon power meter. In addition, we designed a tool to detect ATEBs. This tool utilized

the Android debug bridge commands to extract system related information. The tool effectiveness was evaluated using five Android apps. Compared to power meters, our tool can pinpoint the root cause of the energy bug. Furthermore, it can help in detecting hidden energy bugs when power measurements are not enough alone as in test scenarios 1-3.

For the future, we are planning to fully implement the tool to support other types of app components such as listeners, audio, and wireless services. Also, we are planning to develop light inexpensive models to estimate energy cost for different types of ATEBs. The objective is not to give an accurate estimation of power loss, rather, the idea is to enable the app developer to prioritize fixing ATEBs according to their severity on user experience.

REFERENCES

[1] R. A. Powers, "Batteries for low power electronics," *Proc. IEEE*, vol. 83, no. 4, pp. 687–693, Apr. 1995.

[2] Y. Liu, C. Xu, and S. C. Cheung, "Diagnosing energy efficiency and performance for mobile internetware applications," *IEEE Softw.*, vol. 32, no. 1, pp. 67–75, Jan. 2015.

[3] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *Proc. 10th ACM Workshop Hot Topics Netw.*, p. 5, 2011.

[4] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in Android applications," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2013, pp. 411–420.

[5] 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 *Proc. 10th Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 267–280.

[6] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 588–598.

[7] A. Abbasi *et al.*, "A framework for detecting energy bugs in smartphones," in *Proc. 6th Int. Conf. Netw. Future (NOF)*, Oct. 2015, pp. 1–3.

[8] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying Android Apps for the absence of no-sleep energy bugs," in *Proc. Workshop Power-Aware Comput. Syst.*, 2012, p. 3.

[9] K. Kim and H. Cha, "WakeScope: Runtime wakelock anomaly management scheme for Android platform," in *Proc. Int. Conf. Embedded Softw. (EMSOFT)*, Oct. 2013, pp. 1–10.

[10] A. Banerjee, L. K. Chong, and C. E. A. Ballabriga, "EnergyPatch: Repairing resource leaks to improve energy-efficiency of Android Apps," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 470–490, May 2017.

[11] X. Wang, X. Li, and W. Wen, "WLCleaner: Reducing energy waste caused by wakelock bugs at runtime," in *Proc. IEEE 12th Int. Conf. Dependable, Autonomic Secure Comput. (DASC)*, Aug. 2014, pp. 429–434.

[12] A. Maddahi, W. Kinsner, and N. Sepehri, "Internal leakage detection in electrohydrostatic actuators using multiscale analysis of experimental data," *IEEE Trans. Instrum. Meas.*, vol. 65, no. 12, pp. 2734–2747, Dec. 2016.

[13] A. Abogharaf, R. Palit, K. Naik, and A. Singh, "A methodology for energy performance testing of smartphone applications," in *Proc. 7th Int. Workshop Autom. Softw. Test (AST)*, Jun. 2012, pp. 110–116.

[14] (Jun. 2013). *Wakelocks: Detect no-Sleep Issues in Android* Applications*. [Online]. Available: https://software.intel.com/en-us/android/articles/wakelocks-detect-nosleep-issues-in-android-applications

[15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.

[16] C. Crema, A. Depari, A. Flammini, E. Sisinni, A. Vezzoli, and P. Bellagente, "Virtual respiratory rate sensors: An example of a smartphone-based integrated and multiparametric mhealth gateway," *IEEE Trans. Instrum. Meas.*, vol. 66, no. 9, pp. 2456–2463, Sep. 2017.

[17] P. Nazemzadeh, F. Moro, D. Fontanelli, D. Macii, and L. Palopoli, "Indoor positioning of a robotic walking assistant for large public environments," *IEEE Trans. Instrum. Meas.*, vol. 64, no. 11, pp. 2965–2976, Nov. 2015.

ABDUL MUQTADIR ABBASI received the B.E degree in computer systems engineering from the NED University of Engineering and Technology, Pakistan, in 1992, the M.S. degree in information, network and computer security from NYIT, USA, in 2007, the M.A.Sc. degree in information systems management from the Higher Colleges of Technology Abu Dhabi, United Arab Emirates, in 2014, and the M.A.Sc. degree in electrical and computer engineering from the University of Waterloo, ON, Canada, in 2016, where he is currently pursuing the Ph.D. degree. He served with the Higher Colleges of Technology as a Computer and Network Security Engineer from 1995 to 2014. His research interests include Internet of Things, mobile cloud computing, IT security, and embedded systems.

MUSTAFA AL-TEKREETI received the B.Sc. degree in electrical engineering and the M.Sc. degree in computer and control engineering from the University of Baghdad, Iraq, in 1999 and 2002, respectively. He is currently pursuing the Ph.D. degree with the University of Waterloo, Ontario, Canada. He joined the University of Baghdad, Iraq, as a Faculty Member in 2002. His research interests include computer networks, software testing, and performance evaluation of mobile networked applications.

KSHIRASAGAR NAIK received the B.Sc. degree in engineering from Sambalpur University, India, and the M.Tech. degree from IIT Kharagpur, Kharagpur, the M.Math. degree in computer science from the University of Waterloo, and the Ph.D. degree in electrical and computer engineering from Concordia University, Montreal. He was a Software Developer with Wipro Information Technology Ltd., Bangalore, for three years. He was a Faculty Member with the University of Aizu, Japan, and Carleton University, Ottawa. He is currently a Full Professor with the Department of Electrical and Computer Engineering, University of Waterloo. His research interests include dependable wireless communication, resource allocation in wireless networks, mobile computing, vehicular networks, energy efficiency of smartphones and tablet computers, energy performance testing of mobile apps, communication protocols for smart power grids, and energy performance testing of software systems running on servers. He was a recipient of the Outstanding Performance Award in the Faculty of Engineering at the University of Waterloo in 2013. His book *Software Testing and Quality Assurance: Theory and Practice* (Wiley, 2008) has been adopted as a text in many universities around the world, and he has co-authored a second book *Software Evolution and Maintenance* (Wiley, 2014). He was a Co-Guest Editor of three special issues of the IEEE Journal on Selected Areas in Communications. He was an Associate Editor of the *Journal of Peer-to-Peer Networking and Applications*, the *International Journal of Parallel, Emergent and Distributed System*, and the IEEE Transactions on Parallel and Distributed Systems. He is currently serving as a Regional Editor (America) for the *IEEE Journal of Circuits, Systems, and Computers* and an Associate Editor for the *International Journal of Distributed Sensor Networks*.

AMIYA NAYAK received the Ph.D. degree from Carleton University, Canada, in 1991. He is currently a Professor with the School of Electrical Engineering and Computer Science, University of Ottawa. His research interests include mobile computing, wireless sensor networks, and vehicular ad hoc networks. He has served on the Editorial Board of several journals, including the IEEE Transactions on Parallel and Distributed Systems, the *Journal of Sensor and Actuator Networks*, and the *EURASIP Journal of Wireless Communications and Networking*.

MARZIA ZAMAN received the M.Sc. and Ph.D. degrees in electrical and computer engineering from the Memorial University of Newfoundland, Canada, in 1993 and 1996, respectively. She started her career at Nortel Networks, Ottawa, Canada, in 1996, where she joined the Software Engineering Analysis Laboratory (SEAL) and later joined the Optera Packet Core Project at Nortel as a Software Developer. In addition, she has many years of industry experience as a Researcher and a Software Designer with Accelight Networks, Excelocity, Sanstream Technology, and Cistel Technology. Her research interests include renewable energy, analog and digital control techniques for power converters, wireless communication, embedded systems, machine learning, and software engineering.

PRADEEP SRIVASTAVA received the master's degree in engineering from the Memorial University of Newfoundland, Canada. He has over 35 years of research and development experience in the field of communications, including mobile networks in communication industries such as Nortel networks. He is a co-author in several publications in reputed journals.

• • •