Categorization and Detection of Energy Bugs and Application Tail Energy Bugs in Smartphones

by

Abdul Muqtadir Abbasi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

Abstract

Smartphones are the most ubiquitous and popular hand-held devices because of their rich set of features and wide variety of services. However, their daily use is hampered by their high energy consumption, which forces frequent battery recharging. In smartphones, most energy issues are due to energy bugs (ebugs). These energy bugs are said to exist when smartphone software applications (apps) consume more than expected power while executing or continue to consume energy even after these apps are closed or terminated. Therefore, it is very important to develop energy-efficient applications as these energy bugs severely impact user experience and cause significant user frustration.

In the first half of this thesis, we discuss the problem of energy bugs. To develop frameworks and tools that detect energy bugs, we need to characterize the power consuming behaviour of software. To achieve that, we develop an operational definition for energy bugs that can be easily translated to a procedure to detect energy bugs in smartphones. Furthermore, we integrate the proposed definition with a diagnostic framework to provide a step-by-step procedure for application developers to identify different types of energy bugs. Using the proposed testing framework, developers can investigate the existence of energy bugs especially when apps or platforms evolve. We validate the proposed framework with experiments and real-world energy bug examples. The results show that there are energy bugs across different versions of the same app as well as across different versions of Operating systems running on the same smartphone.

Being software builders and application support providers, software developers should make energy efficient applications for end-users. Thus in the second half of this thesis, we discuss 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. Although, computer hardware and software engineers are involved in developing energy efficient mobile systems, unfortunately, the ultimate energy efficiency depends on the software choices and requirements of the end-user. We investigate multiple scenarios demonstrating that an application can consume energy differently when a user closes the app in four different ways (Home, Back, Swipe-out or Force-stop). This difference in energy consumption is also true when the app has different components such as activity or service with or without wakelocks, thus illustrating the trade-offs that end-users can make for the sake of energy consumption. Although these energy bugs trigger during the execution stage, their effect sometimes remain after closing the app. Borrowing a similar concept of tail energy loss from the field of computer networking, we call the loss of battery power, even after the app is closed or terminated, as application tail energy bug (app-tail-ebug). The diagnostic process begin by measuring any difference in energy consumption of the smartphone before and after closing the app by an external power meter, which clearly establishes the existence or absence of application tail energy bug. To verify, we use system utilities such as Android logging system, logcat, bugreport, dumpstate and dumpsys.

Our ultimate goal is to design a tool as an app running on the device, which can analyse system information and suggest the presence of energy bugs. However, Android has strengthened the security of its OS after KitKat version 4.4, and now superuser access is required to run system level commands. Furthermore, no user app is allowed to access system level information unless the testing app is installed as a system app. Therefore, we run our tool on a desktop PC. In summary, the results of this work can be used by application developers to make implementation level decisions to appreciably improve energy efficiency of software applications on smartphones.

Acknowledgements

First and foremost, I am grateful to Almighty Allah for His countless blessings and for giving me the knowledge and strength to accomplish my research work. I would like to express my deepest and sincere gratitude to my supervisor Dr. Kshi- rasagar Naik for his support, guidance and encouragement throughout my research. I would also like to thank all of my research colleagues, former members of my research group and friends for their help and support. I am also thankful to my committee members, Professor Pin Han Ho and Professor Shi Cao, for their constructive and very helpful feedback and comments on this thesis. Last and by far not least, I am indebted to my parents and family for their unconditional love, continuous support and prayers.

Dedication

This is dedicated to my family members including my parents, wife, kids, brother and sisters.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

According to a report published by International Telecommunication Union (ITU), there are nearly 7 billion mobile subscribers worldwide [6]. Smartphones have become so popular for two reasons: (i) compared to cell phones, smartphones offer richer features and services such as powerful computing capabilities, video conferencing, online surfing, cameras, media players and GPS (global positioning system) navigation units; and (ii) there are thousands of applications (commonly referred to as 'apps') that are available on *app stores*. However, smartphones are constrained by their battery life. Unfortunately, the development in battery technology is not at par with the developments in software and hardware systems [23]. Therefore, paramount attention from both academia and industry has been given for developing more power efficient hardware and software.

Although energy is consumed by the hardware components of the smartphone, this consumption is mainly driven by the software part of the system [24]. Therefore, maximizing the energy efficiency of the software is very important. However, due to high market competition, app developers rarely have sufficient time to carefully optimize their application's energy consumption [15]. Thus, many applications suffer from energy bugs. These energy bugs severely impact the user experience and cause significant user frustration. The term 'energy bug', as the name suggests, attributes that error to the energy of the system. In the presence of an energy bug, the application may not fail/stop but may only cause a higher energy consumption, which makes energy bug very difficult to detect [19]. In literature, there is a lack of useful information that can guide developers to detect the presence of energy bugs in smartphones [17], [18], [19], [11]. Furthermore, without any guided approach, the available energy bug definitions are very broad emphasizing only the causes. Another difficulty is how to distinguish the power consumption of the faulty code from the correct code. There are many factors, such as network conditions, user behaviour and other environmental conditions which can interfere with the test execution and make the power consumption of the same test input different from one run to another run [19].

In smartphones, aggressive sleeping policy is applied, which by default keeps every component in idle state, unless the developer writes code to instruct the OS to keep it on.

List of components broadly fall into two categories.

Traditional Components: CPU, Wi-Fi, NIC, 3G radio, memory, screen and storage that are also found in desktop and laptop machines.

Exotic Components: GPS, camera and various sensors available only in smartphones.

The power consumption of individual I/O components (camera, GPS, etc.) is generally higher than the power consumption of CPU in case of smartphones. Smartphones face a new class of abnormal system behavior, namely energy bugs that causes unexpected amount of high energy drain by the system. Hence the focus is to provide developers with different application level optimizations so as to reduce power consumption, to find e-bugs and their root cause. Android has been chosen among others OSs for its high popularity in the smartphone market and as a promising software framework, which is an open software. Android is also a perfect choice to study the detection of energy bugs and providing optimizations in application design, reduce the energy consumption of network, LCD and CPU because of its easily available multi-vendor support.

1.1 Motivation

There have not been significant improvements in battery technology over the last 25 years. Furthermore there is little evidence that this situation will improve dramatically in next few generations. At the same time, hand-held devices are running more applications with widely varying power consumption characteristics. Hence, managing the battery resource will continue to dominate the list of challenges for some time to continue. Designers of software-hardware platforms for smartphones have added power saving features, allowing developers to manipulate their power consumption based on different functionality. Many software developers generally focus on application functionality rather than efficient power consumption. As a result, many smartphone apps drain relatively more power. Android is the latest trend in mobile OS. Even though, android provides a complete set of application, middleware and Linux kernel for the phone applications developer, it does not utilize several standard kernel features. Hence the need of optimizations and energy efficiency is critical.

1.2 Problem Statement

Some mobile applications use a lot of energy, causing unnecessary drain on the device battery. There are numerous reasons why this occurs and, where it is caused, by bad, ignorant or devious development practice, it can and should be rectified. However, there is no procedure or a tool to detect root causes of unexpected battery drain.

- Apps that run without user's permission. Some applications start automatically when the handset is powered on or do not go to sleep when the user stop using them.
- Excessive use of power-hungry device functions, such as GPS, camera, accelerometer and other sensors, or because apps fail to turn off functions when they are no longer required.
- Apps that keep waking the smartphone from sleep mode or prevent the smartphone going into sleep mode, drain the battery.

1.3 Solution Strategy and Contributions

We suggest a testing framework to check the presence of energy bugs in smartphones. First, we provide an operational definition of an energy bug in order to facilitate the diagnostic process. Then, we present a testing procedure to detect certain classes of energy bugs. The logic behind our procedure is very simple. If the software app is free from defects, the power consumption before and after execution should ideally be the same. We apply this criterion in three scenarios: when the app and platform are fixed, when the app is only updated, and when the platform is only updated. For the first scenario, we check the difference in average power consumption before and after closing the app. For the second scenario, we check the difference in energy consumption of the same app before and after upgrade. For the third scenario, we check the difference in energy consumption of the same app before and after operating system upgrade. Furthermore, we suggest a diagnostic procedure to apply the proposed testing framework. In reality, manifestation of energy bug is the most tedious and laborious job. To simulate a real world environment a testing app where energy bug can be controlled by the developer is developed, furthermore, to verify existence of energy bug, a tool has also been developed to monitor system level activities inside smartphone.

In summary, we make the following three contributions.

- We provide an operational definition of an energy bug and a procedure to check the presence of certain kinds of energy bugs (wakelocks, vacuous background services); and validate our definition by two different experiments as well as by real-world energy bug examples.
- We develop a java based software tool to detect root causes of energy loss when it is not running.
- We prove the correlation between two detection procedures i.e. symptom based verses root-cause based. We find out the root-causes of the battery drain, initially detected as battery depletion symptoms.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In chapter 2, we present the research literature related to energy bugs and energy issues in smartphones. In chapter 3, we present an operational definition and a procedure to detect energy bugs and validate our testing framework with experiments and real energy bugs from the literature. In chapter 4, we present potential causes of application tail energy bugs. In chapter 5, we validate the presence of energy bugs initially through external power meter and then verify it using system logs.

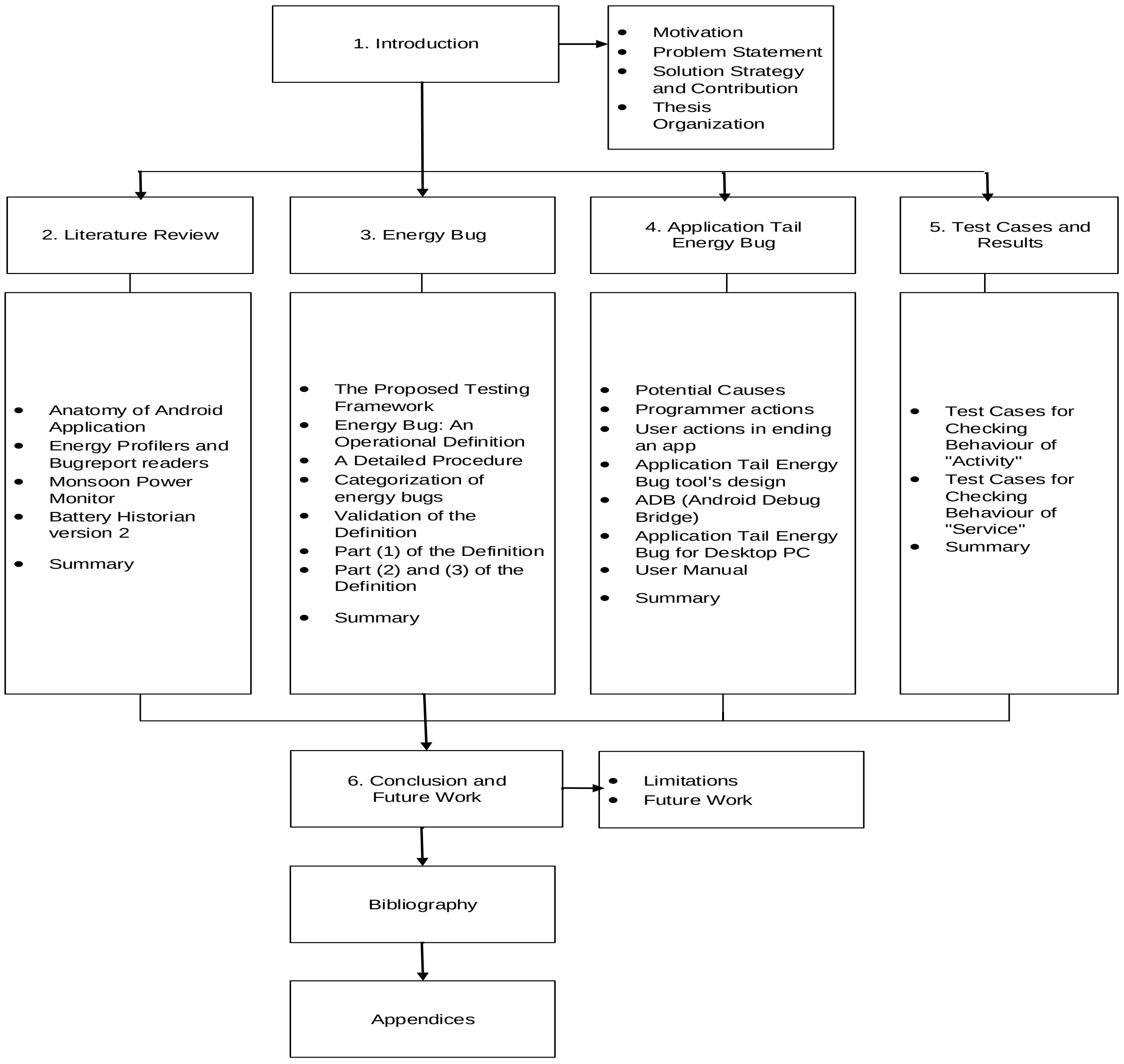

1. Introduction
Motivation
Problem Statement
Solution Strategy and Contribution
Thesis Organization
2. Literature Review
Anatomy of Android Application
Energy Profilers and Bugreport readers
Monsoon Power Monitor
Battery Historian version 2
Summary
3. Energy Bug
The Proposed Testing Framework
Energy Bug: An Operational Definition
A Detailed Procedure
Categorization of energy bugs
Validation of the Definition
Part (1) of the Definition
Part (2) and (3) of the Definition
Summary
4. Application Tail Energy Bug
Potential Causes
Programmer actions
User actions in ending an app
Application Tail Energy Bug tool's design
ADB (Android Debug Bridge)
Application Tail Energy Bug for Desktop PC
User Manual
Summary
5. Test Cases and Results
Test Cases for Checking Behaviour of "Activity"
Test Cases for Checking Behaviour of "Service"
Summary
6. Conclusion and Future Work
Limitations
Future Work
Bibliography
Appendices

Figure 1.1: Thesis organization

Chapter 2

Background

A Sudden battery drain after upgrade of an app or OS in smartphone is referred as energy bug. In order to track energy bugs, first we need to understand anatomy of Android application. We are only interested in software related energy bugs.

2.1 Anatomy of Android Application

An Android application should be thought of as a package comprised of separate, loosely-coupled components connected at runtime, not as a single executable. These components work together by responding to events, or Intents, that are broadcast by other components included in an application, or by the Android system. Each Android app has a manifest file, AndroidManifest.xml, that lists each component that an app includes, and which Intents, if any, will trigger these components to do work. In order to understand the architecture of Android application, Android applications key concepts has to be understood.

- Application components
- Application lifecycle
- Application resources

Android App is basically a Java application executed by a Dalvik Virtual Machine. The Dalvik VM runs .dex files obtained from Java .class files. It is distributed in the form of Android Package(.apk file). It is executed by the Android Operating System (a multi-user Linux) in a sandbox. Each application has a unique user ID and its resources can be accessed only by that user. Each application runs in its own Linux process and its own Virtual Machine instance. Despite this, two different applications can be assigned same user ID in order to share their resources. Critical operations such as access to Internet, read or write contacts, monitor SMS, and access GPS module can be restricted or can require user permission by using the application manifest file (AndroidManifest.xml) An Android application is represented by one or more activities (an activity can be associated with a user screen or window, but usually it is more than that and Linux process. An activity lifecycle is not related to the application process lifecycle. The activity can run even if its process does not exists any more (this is different than C, C++ or classic Java applications where an application has a process). An application can start a component (activity) from another application; the component is executed by its parent application process. An Android application does NOT have a single entry point, like the main() method in Java, C or C++ applications.

Components of Android applications

The most important components of the Android application are:

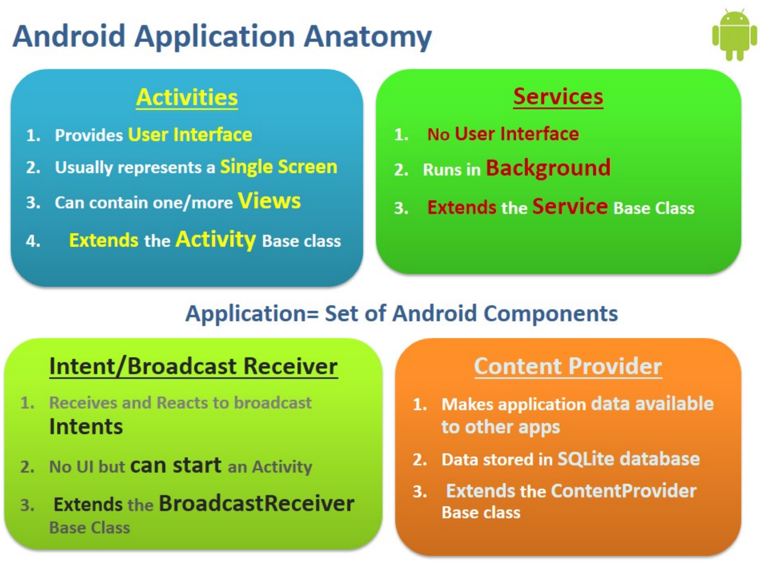

Android Application Anatomy
Activities
1. Provides User Interface
2. Usually represents a Single Screen
3. Can contain one/more Views
4. Extends the Activity Base class
Services
1. No User Interface
2. Runs in Background
3. Extends the Service Base Class
Application= Set of Android Components
Intent/Broadcast Receiver
1. Receives and Reacts to broadcast Intents
2. No UI but can start an Activity
3. Extends the BroadcastReceiver Base Class
Content Provider
1. Makes application data available to other apps
2. Data stored in SQLite database
3. Extends the ContentProvider Base class

Figure 2.1: Android application components

Activity

It represents a user interface screen, window or form. An Android application can have one or more activities. An agenda application can have an activity to manage contacts, an activity to manage meetings and one to edit an agenda entry. Each Activity has its own lifecycle as shown in Figure 2.2 independent from the application process lifecycle. Each Activity has its own state and it is possible to save it and restore it. Activities can be started by different applications (if it is allowed). Has a complex lifecycle because applications can have multiple activities and only one is in the foreground. Using the Activity Manager, the Android System manages a stack of activities which are in different states (starting, running, paused, stopped, and destroyed). In the Android SDK is implemented using a subclass of Activity class which extends the Context class.

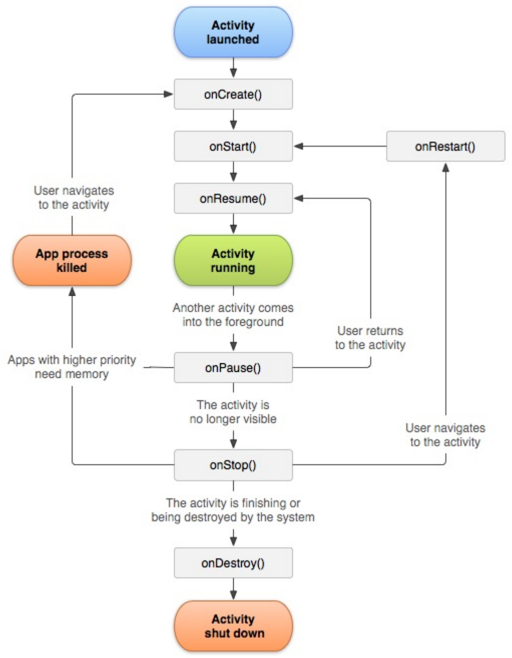

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.2: Android activity lifecycle diagram [7]

Intent

It represents an entity used to describe an operation to be executed. It is a message transmitted to another component in order to announce an operation. Somehow similar to the event-handler concept from .NET or Java. It can considered as an asynchronous message used to activate activities, services and broadcast receivers, implemented by the

Intent class.

Service

A task that runs in the background without the user direct interaction. It is implemented by a subclass of Service. Each Activity has its own lifecycle as shown in Figure 2.3 and 2.4 independent from the application process lifecycle.

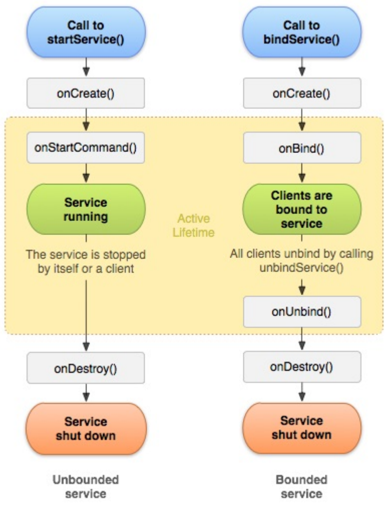


Figure 2.3: Android service lifecycle diagram [8]

Content provider

A custom API used to manage application private data. A data management alternative to the file system, the SQLite database or any other persistent storage location. Implemented as a subclass of ContentProvider. A solution to share and control (with permissions) data transfer between applications (i.e. Android system provides a content provider for the users contact information).

Broadcast receiver

A component that responds to system-wide broadcast announcements. Somehow similar to the global (or system event) handler concept It is Implemented as a subclass of BroadcastReceiver.

Process and threads

When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application

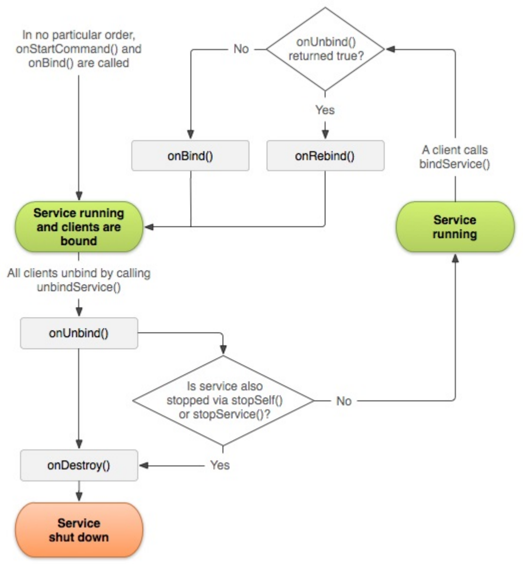

In no particular order, onStartCommand() and onBind() are called
onUnbind() returned true?
No
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
Yes
onDestroy()
Service shut down

Figure 2.4: Android bound service lifecycle diagram [8]

with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread). If an application component starts and there already exists a process for that application (because another component from the application exists), then the component is started within that process and uses the same thread of execution. However, you can arrange for different components in your application to run in separate processes, and you can create additional threads for any process.

When an applications first component starts , the Android system starts a new process for the application with a single thread of execution called main thread or UI Thread. This thread responsible for very important stuff like dispatching events to the user interface widgets. The started Process then may hold Activity, Service or BroadCastReceiver.

All components of the same application usually run in the same process. When new component is started, it will, by default, run in the same thread and inside the application main process unless you provide something else. For example if you run a service by startService() method from an activity, this service runs in the main application process and thread. This is called Single Thread Model.

In this case (running service from an activity to do some intensive work), this single thread model can cause poor performance and may cause your application not responsive and may also see the ANR dialog.

Here comes the need for creating worker thread(s) that runs in background in parallel with the main thread leaving the main UI thread responsive. This is a very common technique that you will do every day when developing Android apps.

2.2 Android Power Management and Wakelock(s)

By default, smartphone are designed to stay in sleep mode. If device is left idle without any touching, first it will go dim, then turn off the display, and finally turn off the CPU. This is how we prevents the device's battery from getting discharged quickly. There are situations when your application might require a different type of behavior:

- Apps for example movie or game apps may need to keep the display turned on.
- Some other apps may not need the display to remain stay on, but in order to do some background jobs they may require the CPU to keep running till it finishes important background jobs.

Android and other mobile OSs have achieved longer battery life by implementing an aggressive power manager. Shortly after use, a phone turns its display off and the CPU goes into a deep power state, thus consuming as little power as possible when not in use. Therefore, phones when they are not used, are able to last several days on a single charge. With Androids power manager, the normal assumption is that when the display is off, so is the CPU. However, Android developers have the freedom to prevent an Android device from going to sleep. They may wish to keep the CPU active, even when the display is off, or perhaps, they want to be able to prevent the display from automatically turning off during a specific activity. For this purpose, Google has included wakelocks in its PowerManager APIs [9] .Applications that want to prevent a device from going to sleep can grab a wakelock. As long as there is an active wakelock on the system, the device cannot enter suspend until the wakelock is released. When using wakelocks, it is extremely important for users to understand that they must properly release wakelocks when they are not needed because an unreleased wakelock will quickly drain a devices battery, by not entering default state to conserve energy. In the case of an app-tail-ebug, we are interested only in the remaining wakelocks not released even after closing the app. These wakelocks were acquired during running the app and should have been released before exiting. Otherwise these remaining wakelocks will keep the device components active and drain the smartphone battery very quickly. The YouTube and Music apps are good examples showing different levels of

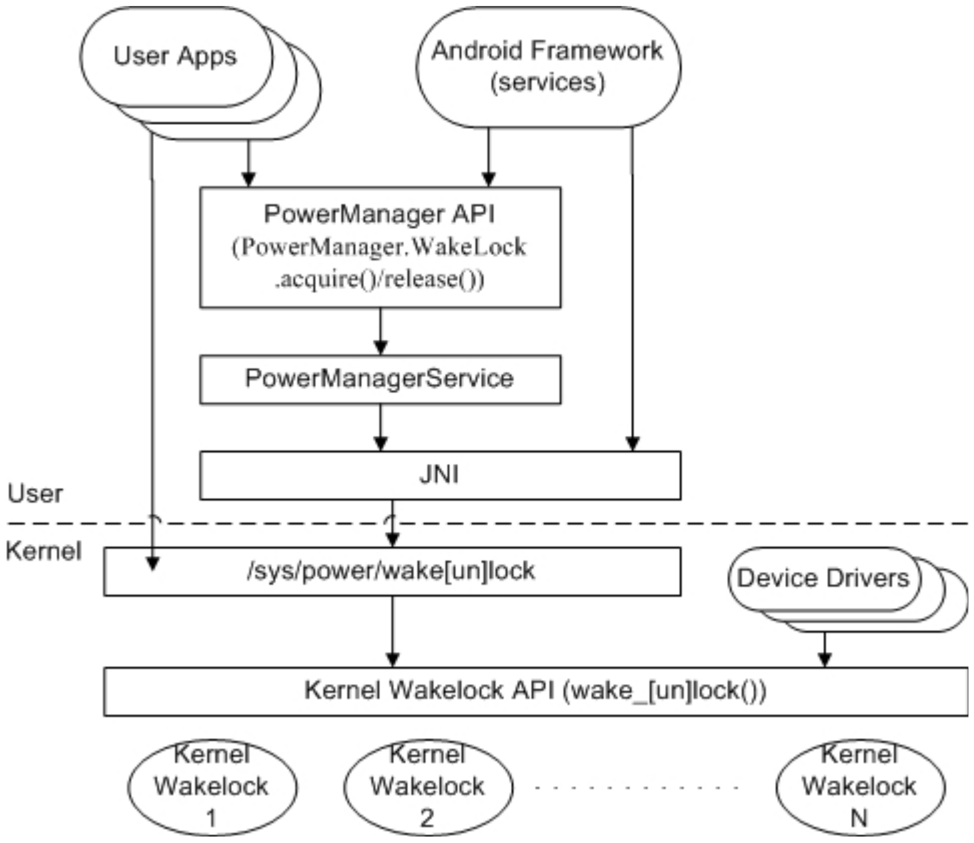


Figure 2.5: Wakelocks overview [3]

wakelocks. The YouTube application will grab a wakelock when the user streams a video. Throughout the duration of the video, the display remains on (ignoring the systems display settings). However, if the user presses the power button during playback, the device will suspend, causing the display to turn off and audio/video playback to cease. In the case of a buggy

YouTube app, remaining unreleased wakelocks can keep the display on indefinitely (ignoring systems display settings). The smartphone will lose its battery earlier than expected. The Music application uses a different wakelock during audio playback. The display settings are not changed, so the devices screen will turn off according to the users display settings. With the display off, the wakelock keeps the CPU in an active state so that audio playback can continue, even if the user presses the power button. Mostly, users wants play music while working on another app. In this case playing voice services in the background indefinitely could eat up smartphone battery silently. In order to stop CPU, programmers should use a timer with the partial wakelock.

Wakelock type	Use	CPU	Screen	Key board
Partial wakelock	acquire or release	Yes	No	No
Screen dim wakelock	acquire or release	Yes	Yes dim	No
Screen bright wakelock	acquire or release	Yes	Yes bright	No
Full wakelock	acquire or release	Yes	Yes bright	Yes

Table 2.1: Wakelock types

2.3 Energy Profilers and Bugs Report Readers

Energy profilers are useful to verify the indication of energy bugs. Though there are many techniques such as collaborative energy debugging. The most accurate tools used by researchers all over the world is Monsoon Power meter. A bugreport is the system state log which is used by all developers for debugging purposes. It is generated and maintained by Android OS. In order to organize it in human readable format, a number of tools has been developed such as CHKBugreport, Battery Historian 2.

2.3.1 Monsoon Power Monitor

The Monsoon Power Monitor, developed by Monsoon Solutions Inc., is a hardware and software-based system that allows the tester to collect power consumption information from any mobile phone that uses a single lithium (Li) battery technology. The hardware component is an external device that acts as a voltage-adjustable power supply, and a current and voltage monitor. The power monitor is connected to the mobile device battery contacts and to a PC to interact with the software component. The software component, PowerTool, is a Windows-PC application that provides a graphical representation of the results and tools to perform basic power calculations. Figure 2.6 and Figure 2.6 show the components of the Monsoon Power Monitor system.

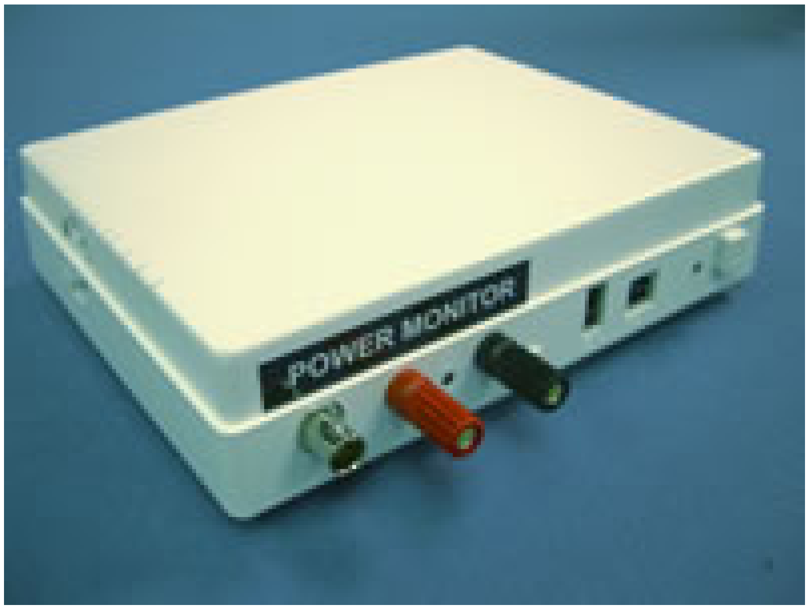


Figure 2.6: Monsoon power monitor

The Monsoon system requires bypassing the phone battery by connecting the phone battery contacts to the Monsoon power supply probes and electrically insulating the terminals of the battery. The approach provides global measurements of power consumption, which means that measurements indicates the overall system energy drainage. This system is used by various testers and researchers for its ease- of-use and accuracy per second, which detect small and short changes in power consumption due to mobile device activities. The limitations of this approach is that it can only be used on mobile devices with accessible

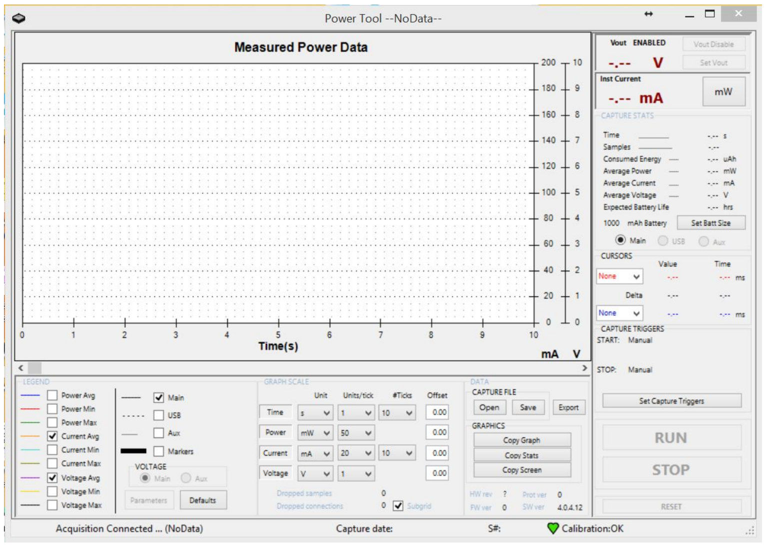


Figure 2.7: PowerTool software graphical tool.

removable batteries. Of the standard approaches, the Monsoon Power Monitor best fits this thesis research needs.

2.3.2 Battery Historian 2

Battery Historian is a tool to inspect battery related information and events on an Android device running Android 5.0 Lollipop (API level 21) and later, while the device was on battery. It allows application developers to visualize system and application level events on a timeline with panning and zooming functionality, easily see various aggregated statistics since the device was last fully charged, and select an application and inspect the metrics that impact battery specific to the chosen application. It also allows an A/B comparison of two bugreports, highlighting differences in key battery related metrics.

2.4 Summary

Before starting a diagnosis process, a complete anatomy of android system has been discussed. it's components and their lifecycle. Energy profilers and bugreport readers are also discussed.

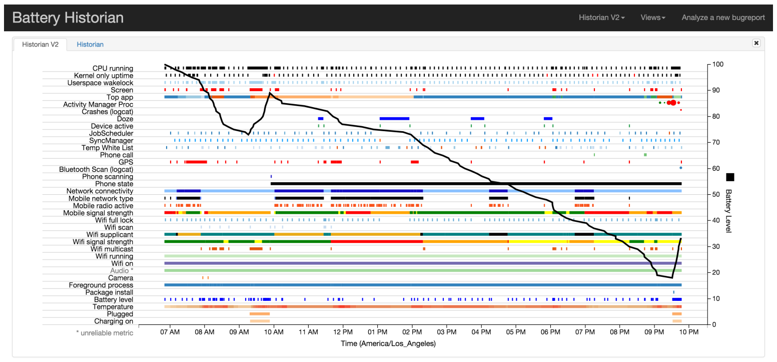
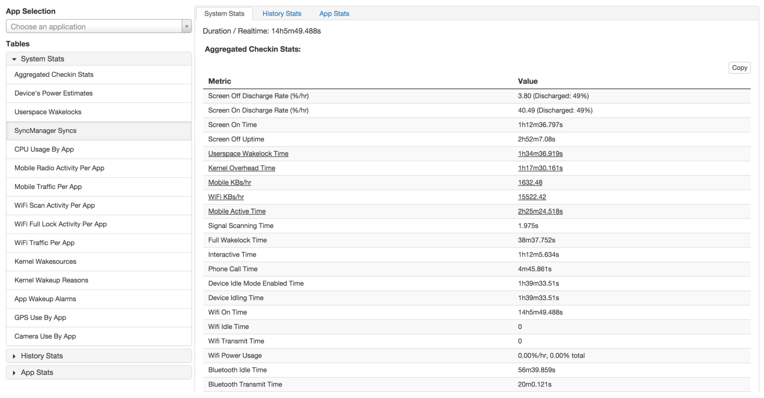


Figure 2.8: Battery historian: Timeline



App Selection

Choose an application

Tables

- System Stats
  - Aggregated Checkin Stats
  - Device's Power Estimates
  - Userspace Wakelocks
  - SyncManager Syncs
  - CPU Usage By App
  - Mobile Radio Activity Per App
  - Mobile Traffic Per App
  - WiFi Scan Activity Per App
  - WiFi Full Lock Activity Per App
  - WiFi Traffic Per App
  - Kernel Wakesources
  - Kernel Wakeup Reasons
  - App Wakeup Alarms
  - GPS Use By App
  - Camera Use By App
- History Stats
- App Stats

System Stats | History Stats | App Stats

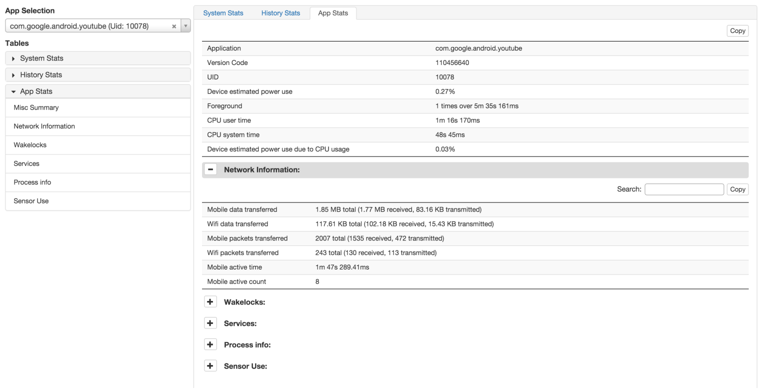
Duration / Realtime: 14h5m49.488s

Aggregated Checkin Stats:

Copy

Metric	Value
Screen Off Discharge Rate (%/hr)	3.80 (Discharged: 49%)
Screen On Discharge Rate (%/hr)	40.49 (Discharged: 49%)
Screen On Time	1h12m36.797s
Screen Off Uptime	2h52m7.08s
Userspace Wakelock Time	1h34m36.919s
Kernel Overhead Time	1h17m30.161s
Mobile KBs/hr	1632.48
WiFi KBs/hr	15522.42
Mobile Active Time	2h25m24.518s
Signal Scanning Time	1.975s
Full Wakelock Time	38m37.752s
Interactive Time	1h12m5.634s
Phone Call Time	4m45.861s
Device Idle Mode Enabled Time	1h39m33.51s
Device Idling Time	1h39m33.51s
Wifi On Time	14h5m49.488s
Wifi Idle Time	0
Wifi Transmit Time	0
Wifi Power Usage	0.00%/hr, 0.00% total
Bluetooth Idle Time	56m39.859s
Bluetooth Transmit Time	20m0.121s

Figure 2.9: Battery historian: System



App Selection

com.google.android.youtube (Uid: 10078)

Tables

- System Stats
- History Stats
- App Stats
  - Misc Summary
  - Network Information
  - Wakelocks
  - Services
  - Process info
  - Sensor Use

System Stats | History Stats | App Stats

Copy

Application	com.google.android.youtube
Version Code	110456640
UID	10078
Device estimated power use	0.27%
Foreground	1 times over 5m 35s 161ms
CPU user time	1m 16s 170ms
CPU system time	48s 45ms
Device estimated power use due to CPU usage	0.03%

− **Network Information:**

Search: Copy

Mobile data transferred	1.85 MB total (1.77 MB received, 83.16 KB transmitted)
Wifi data transferred	117.61 KB total (102.18 KB received, 15.43 KB transmitted)
Mobile packets transferred	2007 total (1535 received, 472 transmitted)
Wifi packets transferred	243 total (130 received, 113 transmitted)
Mobile active time	1m 47s 289.41ms
Mobile active count	8

+ **Wakelocks:**

+ **Services:**

+ **Process info:**

+ **Sensor Use:**

Figure 2.10: Battery historian: App

Chapter 3

Energy Bugs

3.1 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 as shown in Table 3.1. Second, different frameworks are proposed to detect these energy bugs. Finally, to reduce total cost of testing for energy bugs and automate the process, different tools are designed.

The available definitions of energy bugs are very broad, emphasizing only on the causes using non measurable descriptive words as shown in Table 3.1. Even though guidelines to locate energy bugs are provided, the necessary line of actions is not available. Pathak et al. [19] defined and presented a taxonomy of energy bugs in smartphones and discussed the reasons for those energy bugs. An overview of energy bugs by their types is shown in Figure 3.1. We are mainly interested in developing a framework to detect energy bugs that are related to the application and OS (operating system) categories.

In literature, a number of different frameworks have been developed [19], [11]. Our framework is more close to the framework presented in [11]. Pathak et al. [19] proposed guidelines for developing a systematic diagnosis framework to detect energy bugs in smartphones. They recommended to classify the 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. [11] proposed a framework to systematically generate test inputs that help to capture energy bugs. Each test input captures a sequence of user interactions such as touches or taps on the smartphone screen that may cause an energy

Table 3.1: Definitions of energy bugs from literature.

No	Energy bug
a	A sudden and severe battery drain due to mysterious causes including software defects, misconfiguration and environmental conditions [17].
b	An energy bug is a system behaviour that causes unexpectedly heavy use of energy and which is not intrinsic to providing the desired functionality. An app has an energy bug when some running instance of the app (the one in which the bug manifests) drain the battery significantly faster than other instance of the same app (the one in which the bug does not manifest) [18].
c	Ebugs, broadly defined as an error in the system (application, OS, hardware, firmware, external conditions or combination) that causes an unexpected amount of high energy consumption by the system as a whole [22].
d	An energy bug can be described as a scenario where a malfunctioning application prevents the smartphone from becoming idle, even after it has completed execution and there is no user activity. An energy hotspot can be described as a scenario where executing an application causes the smartphone to consume abnormally high amount of battery power even though the utilization of its hardware resources is low [11].

bug in the application. They used a customized version of a third-party tool to generate control flow graphs for event trace generation. That tool does not cover all possible GUI states in the application. Another drawback, the lack of a selection criteria for the suspected user inputs which can stress energy bugs in smartphones.

Regarding tools, there are two main approaches for measuring software power consumption: hardware based and software based. Hardware based measurements use hardware power meters that are wired to the smartphone's battery interface. Software based measurements use software profilers that run on the same smartphone, such as PowerTutor [30], vLens [12], and eProf [20]. Thus, running software profilers on the same platform can interfere with the measurements of the app under test. In hardware based measurements, the measurement devices are not powered by the smartphone's batteries, and, therefore, do not affect the power cost of the smartphone. However, hardware based measurement devices support low sampling rates, while the acquisition rates of the software profilers depend on the frequency of the smartphone processor. Therefore, software based energy profilers are more fine-grained than hardware based measurement power meters [25]. We use a hardware based meter, since our framework follows a black box approach.

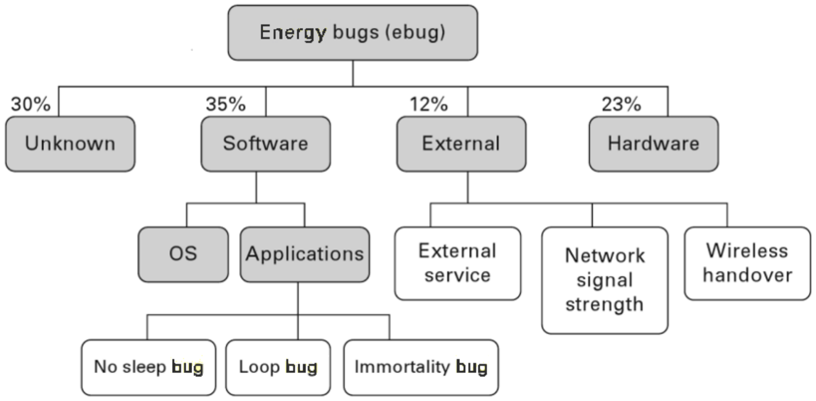


Figure 3.1: Taxonomy of energy bugs [20].

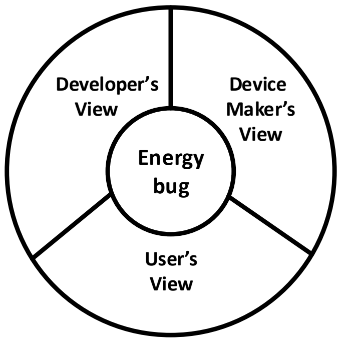


Figure 3.2: Three different views of energy bugs.

3.2 The Proposed Testing Framework

Figure 3.2 shows three different views of energy bugs. Users are mainly concerned about how often they need to charge the battery. Smartphone makers are concerned about the energy consumption of using their platforms. Existing studies show hardware misuse through API (application programming interface) may cause higher energy consumption in smartphones [22]. Developers are more concerned about code level power consumption, where mishandling of resources in coding may cause energy bugs. In this work, we mainly focus on app and OS related energy bugs. In the following section, we present an operational definition and a procedure to detect energy bugs in smartphones.

3.2.1 Energy Bug: An Operational Definition

We define an *energy bug* as follows:

An app is considered to have caused an energy error in a smartphone when at least one of the following conditions holds:

(1) The mean power consumption (P_f) in the smartphone after closing the app is not equal to the mean power consumption (Pi) before opening the app;

(2) The energy cost of app1 is not equal to the energy cost of app2, where app1 and app2 are two different versions of the same app running on the same platform; or

(3) The energy cost of an app running on execution platform1 is not equal to energy cost of the same app running on execution platform2, where platform1 and platform2 are two different versions of the same execution platform.

Accordingly, an energy error is said to be due to an energy bug in the app if the mean power consumption after closing the app is either higher or lower than the mean power consumption before opening the app. The second case indicates that the app has deactivated some features of the smartphone that were active before the app was invoked. The second part of the definition can help developers analyze the behaviour of smartphone when app is evolving. The third part can help developers analyze the behaviour of the smartphone when the platform is evolving. In literature, energy in-efficiencies that manifest during app execution are sometimes called hotspots [11].

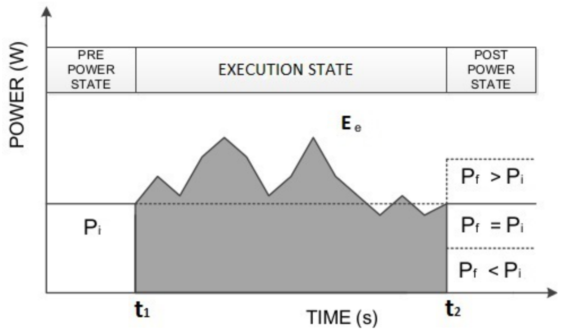


Figure 3.3: Power trace for the application under test.

Figure 3.3 shows a hypothetical power consumption trace of an app. We define three states with respect to the power consumption of the app: *pre-execution state*, *post-execution state* and *during-execution state*. For the pre and post execution 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). For the during-execution state, the app consumption behaviour is characterized by the energy cost (E_e) in joules from time t_1 to t_2. In practice, execution time for app1 and app2 might not be the same. Therefore, we have two scenarios:

Scenario 1: Detection of ebugs after closing the app

According to the definition, a smartphone is in an ideal working condition without the presence of an energy bug, if the pre power state is the same as the post power state. That is:

$$P_f = P_i \tag{3.1}$$

Where P_i and P_f are the pre power state and post power state, respectively. Now, if there is an energy bug in the smartphone, the desired output will not be achieved and the system may be in either of the following two states:

1. **High Power Consuming State**

   In this state, the smartphone consumes on average more power than it was consuming before the execution of the app. It will lead smartphone's battery to drain. Thus:

$$P_f > P_i \tag{3.2}$$

2. **Low Power Consuming State**

   In this state, the smartphone consumes on average less power than it was consuming before the execution of the app. It might affect the operation and performance of the smartphone. Hence:

$$P_f < P_i \tag{3.3}$$

Scenario 2: Detection of ebugs during execution of an app

During the execution of the app, any deviation in power consumption might affect the performance of the smartphone. According to the definition (part 2 and 3), an app would be in an ideal working condition without the presence of an energy bug, if either there is no difference in energy cost when the app is evolving or the operating system is evolving. Hence:

$$E_{e1} = E_{e2} \tag{3.4}$$

Where E_{e1} and E_{e2} are the energy costs of app1 and app2, respectively.

3.2.2 A Detailed Procedure

We integrate the proposed definition in a diagnostic procedure to detect energy bugs, as shown in figures 3.4 and 3.5. In this procedure, we will consider the following types of energy bugs [11]:

- *Resource leak*: resources, such as the WiFi requested by the app during execution, must be released before exiting. Otherwise, they continue to keep the device in a high-power state. We name the set of requested resources as R_r.
- *Wakelock bug*: wakelock is a power management mechanism in Android devices through which applications can indicate that the device needs to stay awake. However, improper usage of wakelocks can cause the device to be stuck in a high-power state even after the application has finished execution. We name the set of requested wakelocks as W_r.
- *Vacuous background services*: in a scenario where an application initiates a service, such as location updates or sensor updates, but does not remove the service explicitly before exiting, the service keeps on reporting data even though no application needs it. We name the set of the initiated services as S_r.
- *Immortality bug*: faulty applications may re-spawn after they have been closed by the user, thereby continuing to consume energy.
- *Suboptimal resource binding*: binding services too early or releasing services too late cause the app to be in a high-power state longer than required. For each resource, we keep two time stamps: the instant at which the resource is binded, and the instant at which the resource is unbinded. We call this set of these two time stamps as T_{srb}.
- *Tail energy bug*: network components tend to linger in a high power state after the workload imposed on them has completed. Note that tail energy does not contribute to any useful work by the component. Scattered usage of network components throughout the application code increases power loss due to tail energy. We call this energy loss as T_e.
- *Expensive background services*: background services, such as sensor updates, can be configured to operate at different sampling rates. Unnecessarily high sampling rates of sensor updates create energy bugs; similarly, fine-grained location updates based on GPS are usually very power intensive and can be replaced by inexpensive, WiFi-based coarse-grained location updates. We name the set of those services as S_{eb}.

- *Loop energy bug*: some portions of application code are repeatedly executed in a loop. For instance, a loop containing network login code may be executed repeatedly due to reasons such as an unreachable server.

3.2.3 Categorization of Energy Bugs

We categorize the above energy bugs into two groups: ebug1 and ebug2. The group ebug1 manifests after closing the app, while the group ebug2 manifest during execution only. The group ebug1 contains resources leaks, wakelock bugs, vacuous background services bugs, immortality bugs. The ebug2 contains suboptimal resource binding bugs, tail energy bugs, expensive background services bugs, and loop energy bugs.

The step by step diagnostic procedure to detect ebug1 is shown in Figure 3.4 and is explained in the following. The step number corresponds to the number associated at the specific box in the flowchart.

Step 1 Before staring the app under test (AUT), measure initial average power consumption P_i.

Step 2 Run the AUT and identify R_r, S_r, and W_r requested by the AUT.

Step 3 After a while, when AUT finishes its job, terminate it.

Step 4 Measure the final average power consumption P_f right after the termination of the AUT.

Step 5 Check if P_f and P_i both are equal.

Step 6 If P_f and P_i are equal, it means no energy bug of type ebug1. Go to the end.

Step 7 If P_f is not equal to P_i, an energy bug may exist.

Step 8 Check if all resources R_r have been released before exiting. These resources were requested by the AUT in the beginning or during runtime. Otherwise, it is an indication of a resource leak energy bug; go to Step 9.

Step 9 Check if all wakelocks W_r have been released before exiting. These wakelocks were requested by the AUT in the beginning or during runtime. Otherwise, it is an indication of a wakelock energy bug; go to Step 10.

Step 10 Check if all services S_r have been released before exiting. These services were requested by the AUT in the beginning or during runtime. Otherwise, it is an indication of vacuous background services related energy bug; go to Step 11.

Step 11 Check if the AUT re-spawn after it has been closed by the user. If yes, it may be an indication of an immortality bug. Otherwise, go to the end.

For Steps 12-16, the app developer can benefit from the already developed methodologies to detect each category of ebug1 [29], [16], [22], [19].

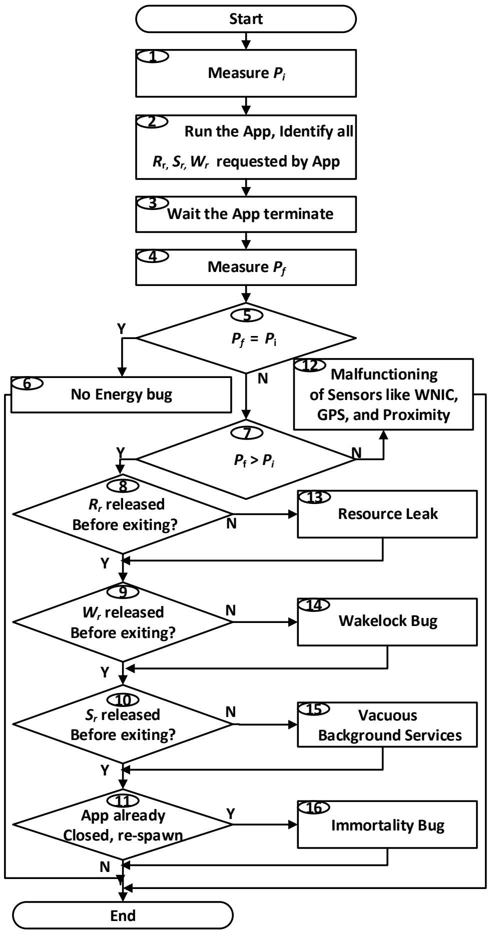

Start
1 Measure P_i
2 Run the App, Identify all R_r, S_r, W_r requested by App
3 Wait the App terminate
4 Measure P_f
5 $P_f = P_i$
Y
N
6 No Energy bug
12 Malfunctioning of Sensors like WNIC, GPS, and Proximity
7 $P_f > P_i$
Y
N
8 R_r released Before exiting?
N
13 Resource Leak
Y
9 W_r released Before exiting?
N
14 Wakelock Bug
Y
10 S_r released Before exiting?
N
15 Vacuous Background Services
Y
11 App already Closed, re-spawn
Y
16 Immortality Bug
N
End

Figure 3.4: Energy bug (ebug1) diagnostic flowchart.

To detect energy bugs type ebug2, we need two applications: app1 and app2. The app1 represents either the pre updated version of the app or the app that runs on the platform1. The app2 represents either the updated version of the app or the app that runs on platform2. The step by step diagnostic procedure to detect ebug2 is shown in Figure 3.5 and is explained in the following. The step number corresponds to the number associated at the specific box in the flowchart.

Step 1 Start measuring the power P_e which is the average power consumption during execution.

Step 2 Start running app1 and label this time instant as time t_1.

Step 3 Identify the set of resources and services that are acquired by the AUT and associate with each of them the time instants at which the services/resources are acquired and released (T_{srb} and S_{eb}). Measure the tail energy T_e.

Step 4 Stop app1 and label this time instant as t_2. Then, stop measuring the power. Do the same Steps (Step 1-4) for app2 and evaluate their energy consumption E_{e1} and E_{e2}.

Step 5 Check if total energy consumption of app1 (E_{e1}) and app2 (E_{e2}) are equal.

Step 6 If E_{e1} and E_{e2} are equal there is no energy bugs; go to the end.

Step 7 If E_{e2} is larger than E_{e1}, there is an energy bug.

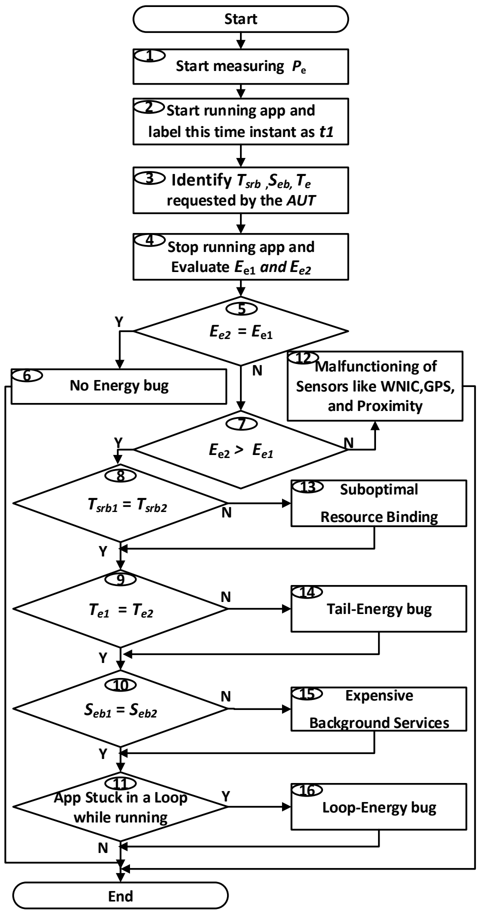
Step 8 Check if suboptimal resource binding T_{srb} are similar for both app1 and app2. It is the timings for binding resources and releasing them. If not similar, it is an indication of suboptimal resource binding type of energy bug. Otherwise, go to Step 9.

Step 9 Check if tail energy T_{e1} of app1 and T_{e2} of app2 are equal. If not, it is an indication of tail energy bug. Otherwise, go to Step 10.

Step 10 Check if expensive back ground services S_{eb1} of app1 and S_{eb2} of app2 are equal, If not, it is an indication of an expensive background services energy bug. Otherwise, go to Step 11.

Step 11 Check if some portions of application code are repeatedly executed in a loop. If yes, it may be an indication of loop energy bug. Otherwise, go to the end.

For Steps 12-16, the app developer can benefit from the already developed methodologies to detect each category of ebug2 [29], [19].



Start
1 Start measuring P_e
2 Start running app and label this time instant as $t1$
3 Identify T_{srb}, S_{eb}, T_e requested by the AUT
4 Stop running app and Evaluate E_{e1} and E_{e2}
5 $E_{e2} = E_{e1}$
6 No Energy bug
7 $E_{e2} > E_{e1}$
8 $T_{srb1} = T_{srb2}$
9 $T_{e1} = T_{e2}$
10 $S_{eb1} = S_{eb2}$
11 App Stuck in a Loop while running
12 Malfunctioning of Sensors like WNIC,GPS, and Proximity
13 Suboptimal Resource Binding
14 Tail-Energy bug
15 Expensive Background Services
16 Loop-Energy bug
Y
N
End

Figure 3.5: Energy bug (ebug2) diagnostic flowchart.

3.3 Validation of the Definition

In this section, we validate the proposed definition through a real example of proximity sensor malfunction. Then, we conduct experiments to prove the presence of energy bugs when apps and OS are evolving.

3.3.1 Part (1) of the Definition.

We validate the first part of the definition by considering real bugs in Apple iPhone and Samsung smartphones. An energy bug that caused a smartphone to consume excessive battery power, eventually leading to battery drainage was detected on iPhone 4S running iOS 5.0.1 and 6.1.1 [1]. iPhone 4S was introduced with a new feature called “Siri”, an intelligent assistant that help the user get things done with voice input. In order to activate “Siri”, users had to press the home button for a long time. However, “Siri” had another option to activate it in the settings called *Raise to Speak*. This option, if activated, allows the users to use “Siri” by just raising the phone to their ears. With this option turned on, the iPhone’s proximity sensor would remain in an active state (consuming more power) for the duration of time this option is on, as the proximity sensor would come to an active state whenever the user receives a phone call or whenever a phone call is made.

Many People reported a problem with the proximity sensor, when “Raise to Speak” is being turned on to activate “Siri”, the smartphone consume more battery power. Ideally, when this option is turned off, the smartphone should go back to its normal power consuming state. That was not the case in iPhone 4S running iOS 5.0.1, whether the “Raise to Speak” option was turned on or off, the proximity sensor was always in an active state. Therefore, the smartphone was constantly in a high power consuming state [2].

As the name suggests, a low power state of the smartphone affects the performance and functionality of the smartphone and leaves the smartphone in a state where it consumes less power than it was before a particular app started. An energy bug that leads the smartphone to this state was detected on Samsung Galaxy Note 3 running Jelly Bean (Android OS v4.3). The issue was again with the malfunctioning of the proximity sensor. Unlike the iPhone case, the proximity sensor malfunction was causing an energy bug to leave the smartphone in a low power state [5].

The energy bug was that whenever a user received a phone call, the proximity sensor being in active state sensed the user’s face and turned the screen off but never turned the screen on again even after the user removed the phone away from his face until the power button was pressed. In an ideal scenario, after a user answers a phone call, the proximity sensor should turn the screen on as soon as the phone is moved away from the face, but due to the presence of this energy bug the phone’s screen remained off, consequently affecting the functionality of the smartphone and leaving the smartphone in a low power consuming state [5].

3.3.2 Part (2) and (3) of the Definition:

For the during-execution state, we observe the actual behaviour of the app. We measure the power consumption of an app in two different scenarios: (a) we observe any changes in power consumption of different versions of the same app running on the same smartphone;

and (b) we observe any changes in power consumption when the same app runs on two different versions of the same OS running on the same smartphone. In the experiments, the app under test is YouTube app for Android devices.

The experimental setup for the validation of our proposed definition of energy bug is shown in Figure 3.6. In order to provide constant voltages, an external power supply is connected to the smartphone and it can support high precision current measurement facility. A laptop computer is used to continuously monitor the power supply and collect the measurement data for the test duration. 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 [9].

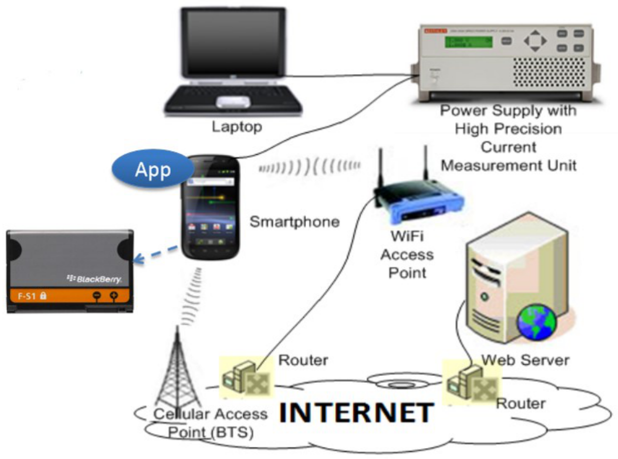

Laptop
Power Supply with High Precision Current Measurement Unit
App
Smartphone
WiFi Access Point
Web Server
Router
Router
Cellular Access Point (BTS)
INTERNET

Figure 3.6: Experimental setup [9].

For consistency in our experiments, we play the same video for four minutes in full screen mode. In order to ensure that the measured energy is the real cost of playing the video, we make sure to clear the cache of the AUT each time the experiment was performed [10]. During an app or operating system upgrade, either there are features being added, bug fixes, or a combination of both. The most important research question (RQ) for the application developer is how to detect energy bugs. In the following section we check the effect of application or OS upgrade on the energy consumption behaviour of the smartphone.

RQ1: (Energy bug detection when app is evolving):
What happens to the power consumption of the smartphone when we use different versions of the same app running on the same platform?

In order to investigate this issue, we use Goggle Nexus S smartphone with a new Android OS image version 4.1.2 (Jelly beans) from google developer's website. We conducted the experiments using the native YouTube app that is installed with the system. Then, we upgrade the app and repeat the same experiment. We perform the same experiment using the same smartphone with the same configurations. Then, we evaluate the percentage of energy difference according to the following relationship:

$$\Delta E_e = (E_{e2} - E_{e1})/E_{e1} \tag{3.5}$$

Table 3.2: Energy costs of different versions of a YouTube app on the same OS.

	App1 running on same OS - Energy (J)	App2 running on same OS - Energy (J)	Difference
Experiment 1	313.04	329.37	4.96 %
Experiment 2	308.44	328.05	5.98 %
Experiment 3	305.58	347.91	12.17 %
Experiment 4	321.67	334.684	6.66 %

Table 4.4 shows the differences in energy consumption by two different versions of the YouTube app. Although app1 and app2 have the same functionality, the power consumption is different. This difference in energy consumption shows that there are energy issues in the app and it may indicate the presence of energy bugs. The results shows that the device is consuming higher energy after upgrade the YouTube app. In other words, we can say that impact of the energy bug has been increased and the device state is getting worse in terms of energy consumption.

RQ2: (Energy bug detection when platform is evolving): *What happens to the power consumption of the smartphone when we use the same app running on different versions of the same platform?*

In order to investigate this issue, we use 'Blackberry Z10' smartphone with a new OS image version 10.2.1. In order to check any deviation in energy cost before and after the upgrade of an OS, we use the native browser app to play same video for six minutes. We perform the same experiment four times before and after the upgrade of the OS using the same smartphone with the same configurations. Then, we evaluate the percentage of energy difference according to the equation 5.

Table 3.3: Energy consumption over different OS versions.

	App running on different OS1 - Energy (J)	App running on different OS2 - Energy (J)	Difference
Experiment 1	569.99	473.88	16.86 %
Experiment 2	564.02	478.91	15.09 %
Experiment 3	564.63	470.18	16.73 %
Experiment 4	568.05	478.04	15.84 %

Table 3.3 shows the results of four different experiments performed on the same smartphone with different OS versions. The results show that different versions of operating systems have different energy consumption patterns. This difference in energy consumption across operating system versions shows the presence of energy bug. The results show that the device is consuming less energy after upgrade the OS. In other words, we can say that impact of the energy bug has been reduced and the device is getting better in terms of energy consumption.

3.4 Summary

In summary, we focused on energy bugs in smartphones. To provide a clear understanding of energy bugs, we developed an operational definition that can be easily translated into a procedure to detect the presence of energy bugs. Furthermore, we integrated the proposed definition in a diagnostic procedure to facilitate the application of the proposed definition. We investigated the existence of energy bugs when apps or platforms are updated. We validated the proposed definition with measurements and realistic energy bug examples. The results showed that there are energy bugs across different versions of the same app as well as across different versions of the same platform.

Chapter 4

Application Tail Energy Bugs

4.1 Related Work

Smartphones are becoming popular because of their rich features and services. However, high energy consumption becomes a bottleneck in the daily use of smartphones. Many efforts have been made by number of researchers to properly manage and reduce the power consumption for smartphone recently, which mainly focus on three aspects: power estimation, energy debugging, and power reduction.[21] [28] [13] [4] As we know, WakeLock is a useful mechanism for developers to keep the system awake when necessary, it is programmer's responsibility to use carefully because misuse of WakeLock will result in unnecessary battery drain quickly. The WakeLock mechanism is provided in the form of a set of APIs in Android developer's documentation. There are many occasions when an acquired WakeLock is not released correctly. This is called a no-sleep bug. With the presence of a no-sleep bug, a large amount of power will be wasted even when the Android device is in the screen-off state, because the device components are not in sleep mode as they should be. Therefore, detecting and correcting WakeLock bugs are very helpful for end users.

Wakelock bugs have has been identified and accepted as main cause of energy loss. In literature, two approaches are very popular for detection purposes, first is static analysis doing a code path scanning, and second is dynamic analysis at run time. Pathak et al. [22] and Vekris et al. [26] applied the first approach by decompiling the Dalvik byte code to Java and then analysing the converted source code statically. However, the problem is it is not easy to decompile all Android packages. When, the static approach lacks information on concrete running states, and this limits its ability to find WakeLock bugs. Kwanghwan and Hojung [14] present a runtime scheme called WakeScope to handle WakeLock bugs when running the AUT. However, user intervention is required, which definitely limits its usage and utility. Another problem is, it can not detect Wakelock bugs when the screen if off, mostly a major reason for wasted battery depletion. Its dependence on Kprobes, a kernel tool, means kernel recompilation is needed at field use, which is infeasible for most end users. Online article [3] that discussed about how to identify WakeLocks with shell command at runtime and performed a simple analysis for WakeLock using BetterBatteryStats application. However, it did not do continue further research for energy saving perceptive.

4.2 Potential Causes

we are adopting the same concept used in networking called tail energy loss. Lingering energy loss after closing the app is a very complex and prominent type of energy bug. We call it application tail energy bug (app-tail-ebug). It is caused by various faults triggered

No App tail ebug	App tail ebug
As per Intended behaviour or user specifications	Any deviation from Intended behaviour or user specifications

Table 4.1: Indication of application tail energy bug

by programmer or user actions. Programmer mistakes in coding create a flaw in the app, which could activate an app-tail-ebug as a result of certain user actions. As the name suggests, application tail energy bug is energy wastage due to some unwanted processing after stopping or closing the app. This energy loss indicates that the app does not allow at least one component of the phone to sleep, resulting in unnecessarily prolonged battery drain. Based on the potential causes Figure 4.1, we have categorized them into two main groups.

- Programmer actions
- User actions

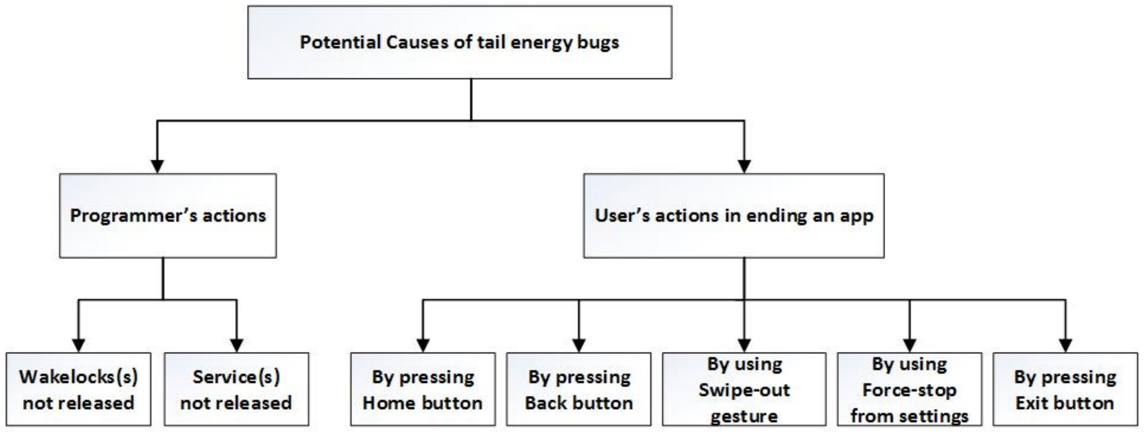

Potential Causes of tail energy bugs
Programmer's actions
User's actions in ending an app
Wakelocks(s) not released
Service(s) not released
By pressing Home button
By pressing Back button
By using Swipe-out gesture
By using Force-stop from settings
By pressing Exit button

Figure 4.1: Potential causes of application tail energy fault

4.3 Programmer Actions

An app-tail-ebug is caused by two main programmer actions. These are coding errors include mishandling of wakelocks and services. All related wakelocks should be released and services should be stopped after closing the app.

4.4 User Actions in Ending an App

There are five main causes of app-tail-ebug caused by users actions. Here we will discuss them one by one. Sometimes specific user's action might leads to app-tail-ebug. The Home button on the virtual task bar may seem to close apps quickly, but, in fact, it actually just minimizes the apps. This means that the app is still technically running (usually in some sort of "paused" state) and that you will be able to resume your app quickly if you try to reload the app. Because the home button does not truly "close" apps, it just gets them out of the way, it is best to avoid using it when you are trying to end processes and conserve your Android device's resources including battery. User can terminate a running app by pressing exit button (if provided) or by swipe it out from the stack or can kill by forced stop from settings. The last option is handled by OS and it will close the app completely including activities and services. Depending on the coding errors behind first two actions might leads to application tail energy bug. We observed a problem with Aripuca GPS app. Location services were supposed to be unbounded when user swipe the background app from the stack but it was not the case. Remaining background location services were continuously running and consuming energy. Another problem has been observed in FM Radio app. Even though user tried to close the app by swiping it out from the stack but voice services were not stopped completely. 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 all related voice services would have been stopped as well. In reality, voice services never stop by itself and drain all battery power in few hours.

The Best way to "Quit" an Android App

You may have noticed that most Android applications do not provide an "Exit" or "Quit" button. This is especially true in the case of all the stock apps produced by Google none of them have any method of directly quitting. It may be a bit disconcerting coming from a desktop-computer environment (e.g., Mac OS, Windows, or desktop Linux), but this is very much intentional and is a fundamental feature of the Android operating system. On Android, it is the job of the Android OS itself to load and unload applications from memory as it sees fit. When you launch an application, the OS will ensure it is in memory. When you stop using the application, the OS will leave it in memory until that space is needed for other purposes. Contrary to the belief of some people who market "auto-killing task managers", this behaviour actually improves the performance of the device and increases battery life. Powered DRAM memory consumes the same amount of power regardless of whether it is caching old applications, storing random garbage or storing pure zeroes. Recently used applications start more quickly if they are still loaded into memory. On the other hand, loading applications into and out of memory takes additional time and processing power, which of course has a negative effect on performance and battery life.

The purpose of "Exit" button.

Some applications may have "Exit" options that simply return you to the device's "home screen". The application will remain in memory until the operating system determines that the memory is needed for other purposes. This is the way that the "Exit" feature works in Aripuca GPS tracker. Other applications actually have "Exit" options with teeth. These applications forcibly terminate the virtual machine in which they are operating. This practice is specifically not recommended by Android engineers and is seen as breaking the API. Nevertheless, many application vendors choose to provide this capability. Google itself do not provide this capability in any of their respective apps.

Recommended way to close Apps in Android

In Android, activities (that is, the part of the app you can see) never run in the background. They can only run (and use battery power) while they are on the screen. The activity stops running regardless of whether you use home or back to leave it. The only difference is what data Android asks the app to save, so neither option is "the right way". It just depends on what you want to do.

Home button

If you use home, Android leaves the app in the same state, so that if you come back to it later (e.g. through the recent apps list), it will still be in the same state you left it: on the same screen, with the same stuff shown. For example, if it is an email app, and you were looking at one email, then it will remember which email that was, and show you the same one. Eventually (after about half an hour), Android concludes that you are not coming back to the app, so it resets this state: next time you start the app, it will start from the front/main screen. To continue the example, the email app will forget which email and folder you were looking at, and show you the inbox.

Back button

If you use back, you're telling Android that you don't want to come back to this view. It'll destroy the information about what you were looking at right away. Next time you start the app, it'll show the front screen (e.g. the inbox). As we know, apps can control the behaviour of the back button: for example, web browsers use it to go back in the browser history. What we have described is the default behaviour of the back button, and developers are urged to keep the behaviour like that to avoid being confusing. Swipe-out gesture It will kill the activity and Service. However, services can re-start immediately with start-sticky option.

Force-stop

A google well known engineer Dianne Hackborn writes: ... (L)ong press on recent tasks to go to app info, and hit force stop there. Force stop is a complete kill of the app – all processes are killed, all services stopped, all notifications removed, all alarms removed, etc. The app is not allowed to launch again until explicitly requested.

Cached background processes

Whichever method you use, Android will leave the app in memory (but not running) for as long as it can. This is to be more efficient. When you come back to the app, if it's still in memory, Android can run it again right away; if it isn't still in memory, then Android has to spend time and energy loading the app from storage again. In old Android versions, apps left in memory in the background this way were included in the list of "running apps". This is a little confusing for users; it makes people think the app is really still running, so newer versions call these apps "cached background processes", to make it clear they're only cached, not running.

Background apps

As we know that activities don't run in the background. So how does your email client check for mail? As well as activities, apps can have services. Services don't have any GUI for you to see or interact with, but they do run in the background. Usually, a service will only run infrequently, such as to check mail once an hour, but it's possible for the app developer to run the service all the time, draining your battery. Leaving an activity with back or home doesn't change how Android treats any services from the same app: the service can continue to run, or be triggered later at a given time (next time the mail check is due).

Desktop	Mobile Phone
Minimize screen	Home button
Back	Back button
Only one component of the application cannot be killed	Swipe-out gesture (only activity can be killed)
End task from task manager	Force-stop from settings
Close or exit	Exit button (rarely available)
Use task manager to release RAM	Not advisable to use task manager because OS manages mobile phone's memory.

Table 4.2: User actions (Desktop vs Mobile phone)

4.5 Application Tail Energy Bugs Tool Design

To track down the root cause of unexpected battery drain in smartphones, a software tool is always demanded since long time. Preferably, a tool which can run on smartphone as an app or as an application on developers PC. For device monitoring at runtime a number of solution are available. To capture system information programmatically, either use specific APIs or ADB commands such as dumpsys, dumpstate and logcat. Due to security reasons, Android has strengthen their OS security by imposing some restrictions. Although dumpsys is a very useful tool that can be used to view the system information on an adb shell, these information cannot be retrieved programmatically by a normal application. Starting from Android Kit Kat version 4.4, no user app is allowed to run dumpsys command to fetch any system related information unless it is installed as a system app. Executing this shell command, through android Runtime.getRuntime().exec() sometimes causes error or unexpected stopping of AUT application under test. When try to write the output to DataOutputStream, it shows

Permission Denial : cannot dump <servicename >.

This is because dumpsys has

android:protectionLevel="signatureOrSystem".

Unless AUT is signed with a platform key or built as a system application, dumpsys information cannot be retrieved programmatically. One has to either root the phone or install the application as a system application or change the protection level of DUMP permission and create a new build.

4.6 Android Debug Bridge (ADB)

Android debug bridge (ADB) is the most powerful tool provided by Google to help developers for debugging and monitoring purposes. In order to check memory dump of system services, dumpsys command can be used which provides wealth of system related information. A number of switches are also available to filter out specific information [27].

To get full list of application and system services directly from Activity Manager, we use

```
> adb shell dumpsys activity service
```

To get full list of wakelock(s) directly from Power Manager, we use

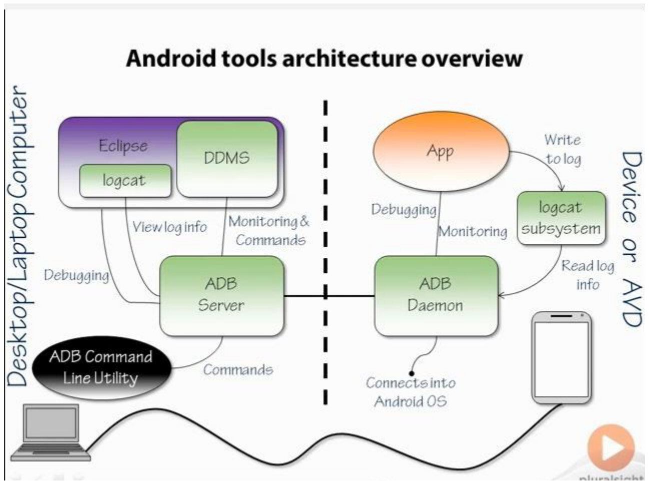


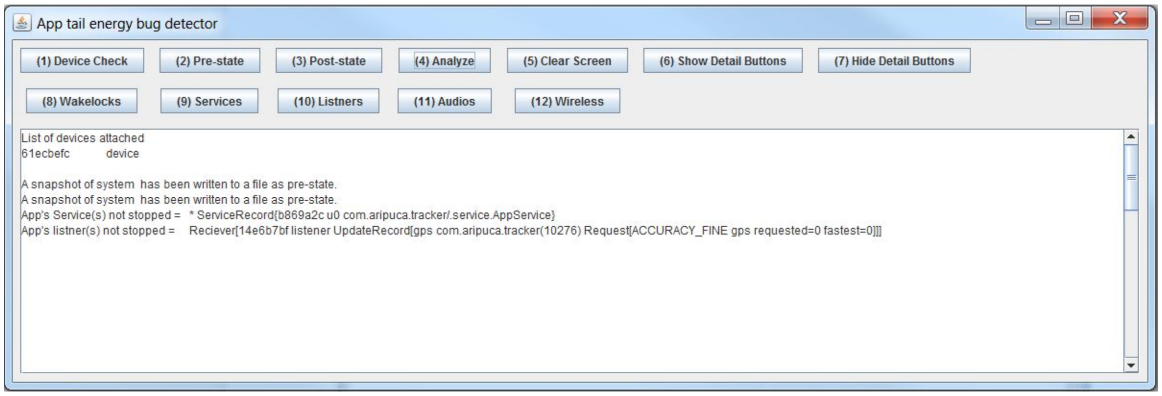
Figure 4.2: Android tools architecture overview

> adb shell dumpsys power

To run ADB commands directly from Java code, a process has been created. Embedded ADB commands in Java code automatically fetch system information from the device. The output is saved in a text file and also displayed in text output area of the tool. The functionality of two button such as pre-state and post-state are identical. First button saves memory dump system services to pre-state text file and second button in post-state file. Later, other buttons such as wakelocks and services are used to compare pre-state and post state and display any difference. The main windows of app-tail ebug tool has two sections. The upper section shows a button palate. The lower section where tools output is displayed.

4.7 Challenges

To avoid security restrictions, a software tool has been developed in java for developers PC as shown in Figure 4.3. The main purpose of this tools is to help developers to save their time, especially while coding an app, sometimes developer makes some potential mistakes which can lead to battery drain at run time. The root cause could be a wakelock not released or a service not stopped before exiting the app. This tool can communicate with Android framework system services and fetch necessary information twice, before and after closing the AUT. Later, by comparing both system state files, it can display remaining wakelocks and services causing battery drain. The technology used behind this tool is a set of management tools and commands.



```
List of devices attached
61ecbefc        device

A snapshot of system  has been written to a file as pre-state.
A snapshot of system  has been written to a file as pre-state.
App's Service(s) not stopped =   * ServiceRecord{b869a2c u0 com.aripuca.tracker/.service.AppService}
App's listner(s) not stopped =    Reciever[14e6b7bf listener UpdateRecord[gps com.aripuca.tracker(10276) Request[ACCURACY_FINE gps requested=0 fastest=0]]]
```

Figure 4.3: Application tail energy bug tool

Methodology:

In each experiment, first we check an indication of app tail energy bug using Monsoon power meter. If power consumption after closing the app is higher than before starting the app then it is an indication of app tail energy bug. Once it is confirmed then we verify it using our tool app tail energy bug tool. This tool compares pre and post state memory dumps and show root cause of the problem.

4.8 User Manual

The step by step diagnostic procedure to detect application tail energy ebug is shown in Figure 4.6 and is explained in the following. The step number corresponds to the number associated at the specific box in the flowchart.

Step by Step Procedure:

Step 1 Connect the smartphone through a USB or Wi-Fi to the desktop computer.

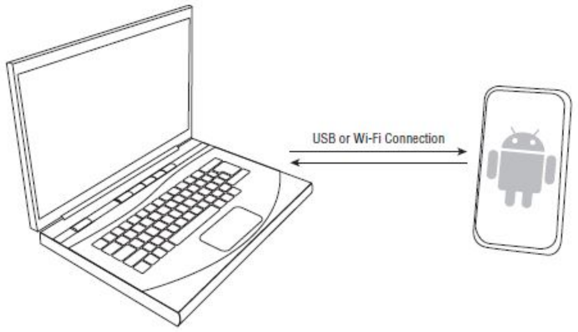


Figure 4.4: Running “Adb” through USB or over Wi-Fi connection

Step 2 Click on Device check button to verify connectivity between smartphone and a PC.

Output:

List of devices attached

61ecbefc device

Step 3 Click on Pre-state button. It captures memory dump of system services and save it in pre-state file, This file contains a list of currently running services and acquired wakelocks, listeners, audio services and Wi-Fi status.

Output:

A snapshot of system has been written to a file as pre-state.

Step 4 Click on AUT icon to start it and then play different features.

Step 5 Close the AUT by pressing HOME button.

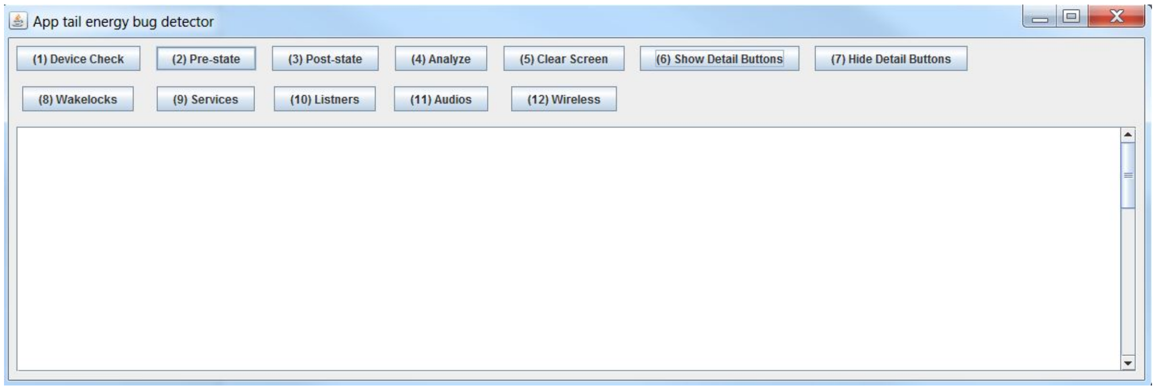


Figure 4.5: Application tail energy bug tool

Step 6 Click on Post-state button. It captures memory dump of system services and save it in pre-state file, This file contains a list of currently running services and acquired wakelocks, listeners, audios and Wi-Fi connections.

Output:

A snapshot of system has been written to a file as post-state.

Step 7 Click on Analyze to compare both files pre-state and post-state files line by line.

Step 8 If both files are equal, it means no app tail ebug go to step 9. Otherwise show list of remaining wakelocks, services, listeners, audios, Wi-Fi connections etc.

Output:

List of devices attached

61ecbefc device

A snapshot of system has been written to a file as pre-state.

A snapshot of system has been written to a file as post-state.

App's Service(s) not stopped = * ServiceRecord2e7e353f u0 com.aripuca.tracker/.service.AppServic

Click on Clear button to clean the output window.

To show or hide details buttons, click on show or hide details. A palette of details like wakelocks, services, listeners,audios and Wi-Fi button will be displayed.

Step 10 If both files are not equal, it means app has stopped some components which were active before starting AUT.

Step 11 Click on Wakelock not released to compare both files prestate-Wakelocks and poststate-Wakelocks files line by line. If both files are equal, it means no app tail ebug. Otherwise show list of remaining wakelocks.

Step 12 Click on Services not released to compare files prestate-Services and poststate-Services line by line. If both files are equal, it means no app tail ebug. Otherwise show list of remaining services.

[Step 12][Step 13][Step 14] are the result of the experiments.

Algorithm: Detecting Application tail energy bug (A parser)

Input:

pre-state and post-state memory dumps of system services such as activity, power, location, Wi-Fi etc. services captured before starting and after closing the AUT.

Output:

List of remaining wakelocks, services, listeners, Wi-Fi connections after closing the AUT. These were acquired/started during execution of AUT.

Body:

```
Initialize: line1, line2
While (line2 != null)
    While (lin1 != null)
        if line2 = line1 then
        print no indication of app-tail-ebug
    else if line2 != line1 then
        print indication of app tail-ebug
        print list of remaining wakelocks and services
    end
end
```

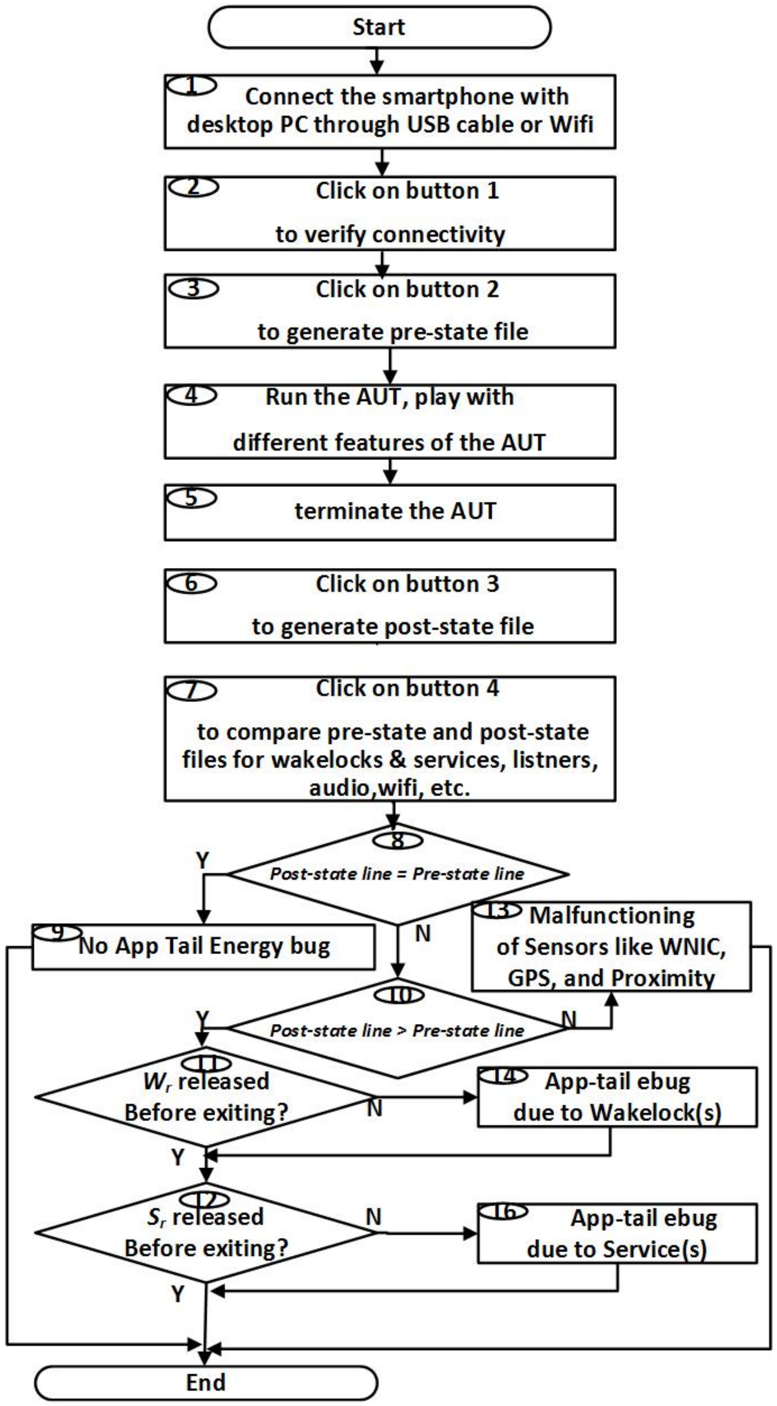

Start
1 Connect the smartphone with desktop PC through USB cable or Wifi
2 Click on button 1 to verify connectivity
3 Click on button 2 to generate pre-state file
4 Run the AUT, play with different features of the AUT
5 terminate the AUT
6 Click on button 3 to generate post-state file
7 Click on button 4 to compare pre-state and post-state files for wakelocks & services, listners, audio,wifi, etc.
8 Post-state line = Pre-state line
Y
N
9 No App Tail Energy bug
13 Malfunctioning of Sensors like WNIC, GPS, and Proximity
10 Post-state line > Pre-state line
Y
N
11 W_r released Before exiting?
N
14 App-tail ebug due to Wakelock(s)
Y
12 S_r released Before exiting?
N
16 App-tail ebug due to Service(s)
Y
End

Figure 4.6: Overall testing procedure

4.9 Wakelocks and Services Behaviour

Objective: To check the behaviour of an application when acquired wakelock(s) is not released before exiting.

As the name shows, wakelock is a combination of two words i.e. wake (out of sleep) and lock (stay). The overall meaning of wakelocks means stay out of sleep or stays awake. As compare to desktops, mobile phones are designed to stay OFF to conserve batter power. Because a desktop has a permanent power connection that is why, by default they are designed to stay ON. The dictionary meaning of mobile is something not stationary. Smartphoes are mobile devices. Though, these devices do not have a permanent connection to the power source but have a limited size battery with wireless connectivity available. In order control power consumption of the device Google has provided a mechanism for software developers so that they can keep the device ON whenever they want to perform some useful task. These software controlled power switches are named as Wakelocks. Currently, developers have full control on three resources CPU, display and keyboard in group format. There are four wakelock groups. Only partial wakelock allows to acquire wakelock on CPU individually. The Wakelock forces the device to stay in awake state. Battery depletion rate depends upon the type of wakelock acquired or load such as CPU, display and keyboard. Each wakelock provides a control on different combination of resources. As shown in table1, depending upon the number of components, full wake lock consumes highest battery power due to all components (CPU, display and keyboards) are ON, second highest is screen bright wakelock where (CPU, display) are fully ON, the third is screen dim wakelock where (CPU, display) are on but screen is dim and lastly partial wakelocks where only CPU is ON. Indication of wakelock related issues in mobile devices: First, a user can configures his smartphone display time out, let us say, just one minute. If no one touches the mobile phone for one minute, ideally display should go off after one minute time out and after few seconds CPU should also go into deep sleep. However, In case of last three wakelocks in table1, screen and CPU will never go to sleep. Screen will either stay fully on or in dim state. In case of a partial wakelock, screen will be off but devices will be losing battery power due to CPU which will be running silently. Note: Wake lock will be triggered after display timeout.

Experimental Setup:

Following items are the main components of our work bench.

- A developer computer (desktop or laptop)
- A smartphone with android OS (Samsung Note with android ver. 5.11)
- A monsoon power monitor device for measuring power consumption
- A router for wireless connectivity to run ADB for fetching bugreport

System Model:

(Recommended: option1)

Option1: (Developers PC and Smartphone are connected through a wireless router)

Option2: (Developers PC and Smartphone are connected through a USB cable. A persistant partial wakelock acquired by the system would not allow accurate power consumption.)

Step by Step Procedure:

Step 1 By pass the smartphone battery and connect it monsoon power meter for steady power.

Step 2 Run monsoon power meter on desktop PC.

Step 3 Click on vout button from right top corner of power tools window to enable output voltages.

Step 4 Start smartphone

Step 5 Connect smartphone with developers PC through USB cable.

Step 6 Run adb from command prompt

Step 7 Check device connectivity using command (adb devices) you can see (List of devices attached)

Step 8 In order to switch over to Wi-Fi, use following command (adb tcpip 5555, adb connect IP address of device)

Step 9 Once connectivity is established, remove the USB cable. To re-confirm device connectivity repeat step 2.

Step 10 Set the display time to 2 minutes.

Step 11 Run app tail ebug tool on developers pc.

Step 12 Do not touch smartphone. Let the smartphone go to sleep.

Step 13 Start measuring power consumption for two minutes. In the meantime, click on pre-state button on app tail ebug tool to generate pre-state bug report.

Step 14 Start the smartphone by clicking on Home button. Continue measuring power consumption.

Step 15 Let the device running. Continue measuring power consumption and do not touch it for two minutes.

Step 16 Click on AUT (App under test) to run it. Click on All Wakelocks button from AUT. Continue measuring power consumption and do not touch it.

Step 17 After two minutes, stop the AUT by clicking on Home button. Continue measuring power consumption and do not touch it.

Step 18 As per display time out set earlier, after two minutes screen will turn to dim and then completely dark. Device will go to sleep state or maybe not depending up the existence of wakelocks. Continue measuring power consumption.

Step 19 After a threshold time of 30 seconds take another bugreport by clicking on post-state button on app tail ebug tool.

Step 20 Continue measuring & monitoring power consumption for a while in order to check if device goes to sleep state or not at all.

Step 21 Click on wakelocks button on app tail ebug tool to analyze pre-state and post-state bugreports

Step 22 To verify manually if any wakelock is not released after stopping the AUT look between Wake Locks: size= and Suspend Blockers: size= in post-state bugreports.

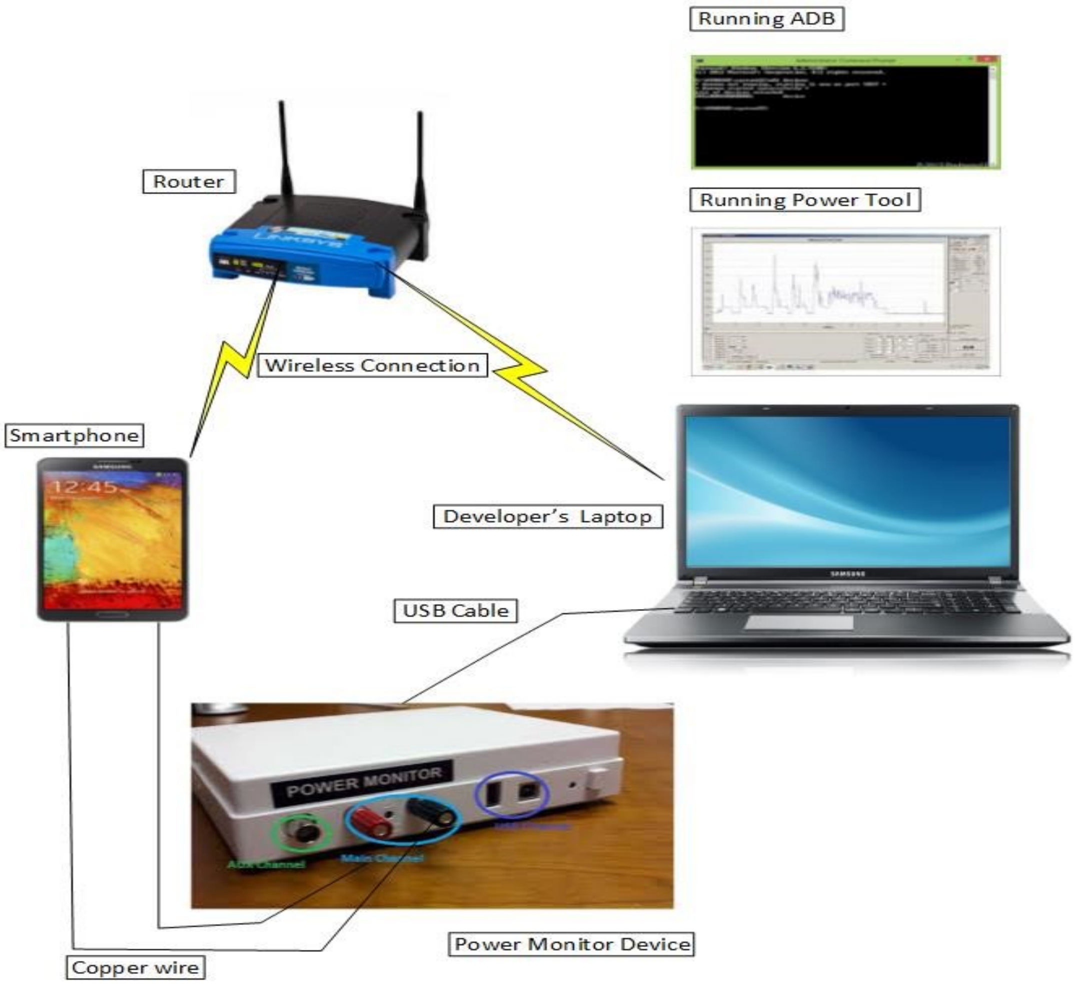


Figure 4.7: System model

4.10 Wakelock Demo App (App Under Test)

In order to verify our assumption about wakelock-related problems, we have developed a wakelock demo app for Android as shown in Figure 4.8. This app has two screens. The first one is Activity screen which is designed to check activity behaviour with or without wakelocks provides options for four types of wakelocks i.e. partial wakelock,screen dim wakelock, screen bright wakelock and full wakelock. There are two buttons provided next to each wakelock type. The ON button is used for acquiring and OFF button for releasing the wakelock. Another button is provided to change the screen. The second screen is Services screen, whcih designed to check services behaviour with or without wakelock. Four types of wakelock mentioned earlier has been added in the coding. There are two buttons provided next to each wakelock type. The Start button is used for starting and stop button for stopping the service. Another button is provided to start a browser app.

The purpose of this experiment is to check the power consumption behaviour of the AUT especially once the app is not visible. As discussed section 2.1.1, wakelocks are software switches to control CPU, display and keyboard. It is very important to keep in mind that once a wakelock is acquired it has to be released by the programmer. Android system will never release them unless a process is killed or devices is re-started. Some abbreviation used in Figure 4.8.

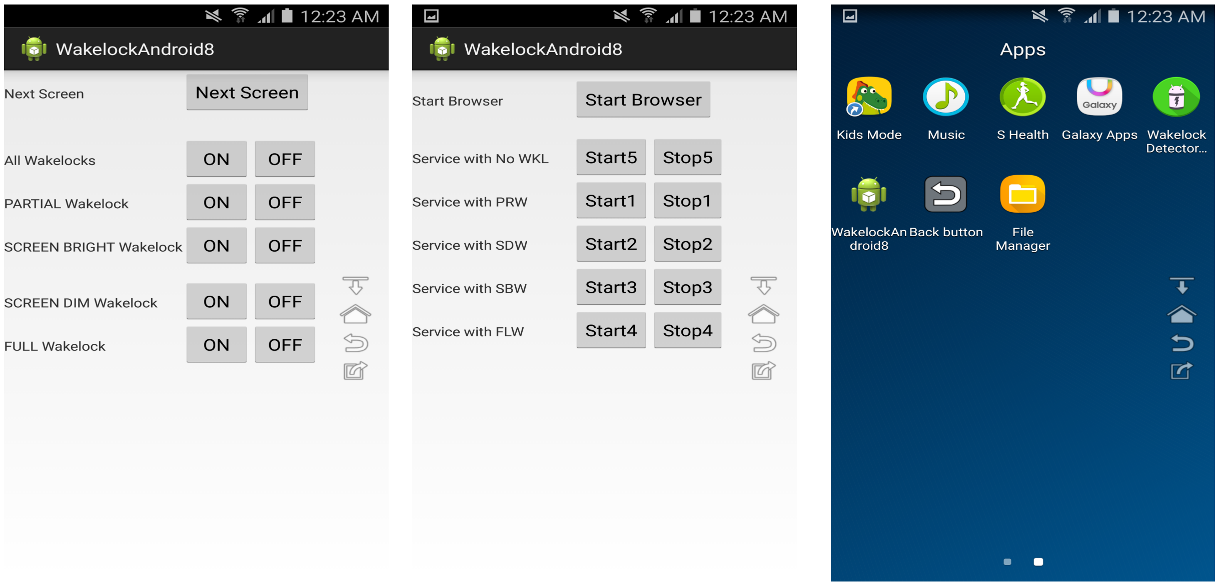
PWR = Partial wakelock
SDW = Screen dim wakelock
SBW = Screen bright wakelock
FWL = Full wakelock
WKL = Wakelock

4.11 Quantification of Energy Loss

Application tail energy loss can be classified into two categories.

- Energy loss before display time out.
- Persistent energy loss due to wakelock after screen-off.

In the first scenario; depending upon their convenience, users setup their smartphone's display time out to save power. It could be from 15 seconds to 30 minutes. Device is



(a) Wakelocks Screen (b) Services Screen (c) Home Screen

Figure 4.8: Wakelock demo app

considered fully awake when the display is ON. Background services are allowed to run continuously without any interruption if the display is kept ON. The loss of energy during display time out can be calculated as the product of power and display time out. In second scenario; any unreleased wakelock could keep the device awake indefinitely. Total energy loss will be product of power and total time including display time out.

4.12 Summary

Lingering energy loss after closing the app is a very complex and prominent type of energy bug. We call it application tail energy bug. We discussed about potential causes of application tail energy bug. We categorized them into two main groups. Programmer's and User's actions. Programmer's mistakes in coding such as misuse of services and wakelock are the main reasons. Then, we developed a testing app and a tool to capture programming mistakes in testing at run time.

Chapter 5

Test Cases and Validations

In order to check power consumption behaviour of android application components such as activity and service we have performed various experiments. An activity is the essential component of an android app. We have checked power consumption behaviour of an activity with or without wakelock. Our result showed that activity only runs in foreground and consume battery power. However, activity never runs in background or consume any power. The main advantage of using wakelock cannot help them. Wakelocks has to be released by the programmer once the intended task has been completed.

We also tested services. Our results showed that service can run in the background continuously unless it is not stopped. However it pauses when device goes into sleep mode after display time out. The ability of service to re-launch itself to continue unfinished task makes it power hungry nature.

Services are more susceptible to be killed by the system in case of less resource than activity. Only those services will not be killed which are bound to UI activities. In the following section we will present 20 test case for each components activity and services.

5.1 Test Cases for "Activity"

Experiment # 1A (Wakelock type : PRW, Ending type : Home button)

Objective: To monitor systems behaviour when a partial wakelock is acquired by the AUT but not released before exiting due to specific user action using Home button.

Hypothesis: In this experiment, we acquire a partial wakelock but not release it deliberately to simulate a real life environment. Then, we start and play with another app such as browser app for a while. Later, we come back to the wakelock demo app and close the app using Home button. As per default behaviour of Home button, current screen (activity) should be saved in the memory and then Home screen should be displayed. By default Android system does not take any action to release previously acquired wakelocks. To prove this assumption, we are doing following experiment.

Procedure: In order to check any deviation from ideal behaviour, we have run two test cases. In the Figure 5.1, the dotted line curve represents the power consumption of the app exhibiting buggy behaviour and the solid-line curve represents that of correct behaviour of the app. In the beginning of the test, the smartphone is in the switched off state. We start the phone by pressing the power button. After successful boot up, the smartphone Home screen is completely visible. We do not touch the smartphones screen, and it remains in the same state for a while. At this stage, the device is in the idle state. Due to the aggressive sleeping policy implemented by the Android platform, the device goes into the sleeping mode. At this moment, we start to measure the power consumption of the device which corresponds to P1. We also click on pre-state button on app tail ebug tool on developers PC. Here, the device power consumption is 100 times less than the consumption in the On state; In order to run the test case, we press the power button of the smartphone to wake it up at time T1. The power consumption shots up from P1 to P3. Now, the display is visible, and other important components such as the CPU, display, keyboard and radio are also in full wake up state. At this stage, no app is running on the smartphone. We measured the power consumption P3 during time interval (T1-T2), which corresponds to the app pre-execution state. At time T2, we start the AUT by clicking on the app icon. The power consumption of the device shots up from P3 to P4. The wakelock demo app is loaded and it starts running with full functionality. Now, the app is in the execution state. We test different features of the application. At time T3 we press ON button next to the partial wake lock on WakelockAndriod screen to acquire that particular wakelock. We measure the power consumption P4 during the time interval (T2-T3-T4), which corresponds to the power consumption of the app during execution. In the mean time we also run a browser app and later come back to wakelock demo app. At time T4, the app is stopped by pressing

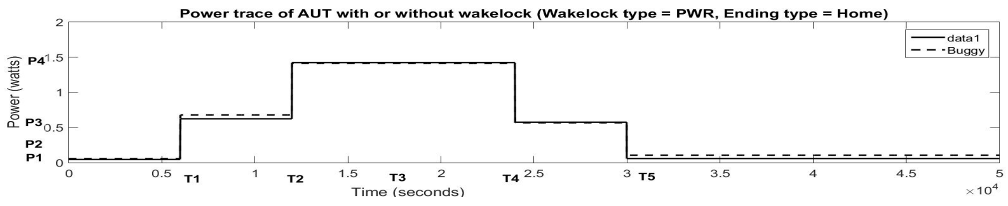
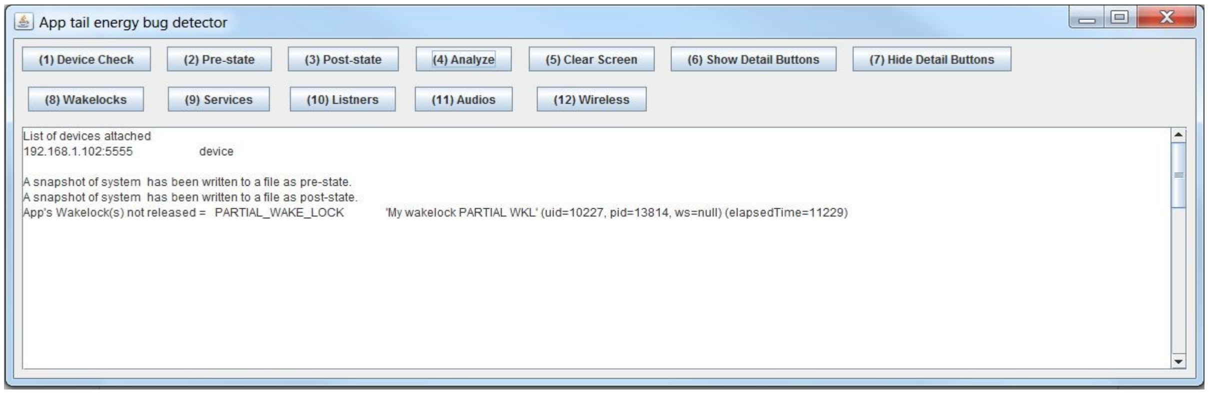


Figure 5.1: Power trace comparison of wakelock app (Experiment # 1A)



```
List of devices attached
192.168.1.102:5555          device

A snapshot of system  has been written to a file as pre-state.
A snapshot of system  has been written to a file as post-state.
App's Wakelock(s) not released =   PARTIAL_WAKE_LOCK         'My wakelock PARTIAL WKL' (uid=10227, pid=13814, ws=null) (elapsedTime=11229)
```

Figure 5.2: Result from application tail energy bugs tool (Experiment # 1A)

the Home button from the WakelockAndriod screen, as shown in Figure 1.3. We do not touch the screen. At time T5, system goes to sleeping mode after screen time out. In our case we setup screen time out two minutes (120 seconds). For simplicity, all steps mentioned above are summarized in Table 5.3.

Starting from T5, the power consumption of the app in the absence of wakelock is completely differs from that of the presence of wakelock. In the absence of wakelock, the app power consumption has dropped from P3 to P1. Once the foreground screen is moved to the background, the AUT should be stopped and wakelock should be released. In contrast, in the presence of the wakelock, the app power consumption did not drop to P1 level and stayed on the P2 level. After the T4, the app is considered to be in the post-execution state. We measure the power consumption during the time interval (T4-T5) and call it the post-execution power consumption of the app. After stopping, we do not touch the device at all. After display time out at T5, the power consumption of the app without a wakelock drops from P3 to P1, whereas with wakelock, it drops from P3 to P2.

Result from App Tail ebug tool: It shows partial wakelock acquired by the AUT has not been released.

Observations: As shown in Figure 5.1, the graph shows that power consumption is always settles down to the pre-execution power state level after stopping the AUT. The device goes to suspend mode after a while when wakelock is not present, whereas in case of a buggy behaviour, wakelock does not allow the device to go into the sleep mode.

As shown in Figure 5.2, the results shows from app tail ebug tool also supports our theory mentioned above. Our tool successfully detects root cause of the problem. A partial wakelock is causing the loss of battery power in the buggy app.

Conclusion: We conducted an experiment to show the presence of the app tail energy bug when a partial wakelocks is not handled properly. We simulated the real environment by using a wakelock demo app. We acquired a partial wakelock from AUT but did not release it deliberately to check its affects after closing the app. While playing with AUT, we switched to a browser app and then come to AUT after a short while. We used Home button to close the AUT. After a display timeout of 120 seconds, screen turned to dim and then completely dark. Partial wakelock keeps the CPU awake but nothing to do with display. The continuous power consumption after display is off shows the presence of partial wakelock which triggered at right time. Using Home button for exiting do not release wakelock(s) acquired by the AUT.

Time	**User/developer actions**	**Device state**	**Comments on power changes**		**App state**
0-T1	User do not touch the smartphone screen for a while. Press pre-state button on app-tail-ebug tool	suspend	Power consumption remains at P1		sleep
T1-T2	At time T1, wake up the smartphone by pressing power button	Idle	At time T1 power consumption shots up from P1 to P3		Pre-execution
T2-T3	At time T2, Click on app icon to run it	On	At time T2 power consumption shots up from P3 to P4		Execution
T3-T4	Press ON button OFF button Next to wakelock type Start browser	On	Correct app behaviour without WKL	At time T3, power consumption changes due to browser app	Execution
	Press ON button Next to wakelock type Start browser	On	Buggy app behaviour with WKL	At time T3, power consumption changes due to browser app	
T4-T5	At time T4, Stop the AUT by pressing Home button	Idle	At time T4 power consumption drops from P4 to P3 in both cases		Post-execution
T5-T6	User do not touch the smartphone screen for a while. Press Post-state button on app-tail-ebug tool	Suspend	Correct app behaviour without WKL	At time T5, power consumption drops from P3 to P1,	Sleep
		Idle	Buggy app behaviour with WKL	At time T5, power consumption drops from P3 to P2,	Awake

Figure 5.3: Summary of the main states of the AUT during test case execution.

Experiment # 2A (Wakelock type : PRW, Ending type : Back button)

Objective: Same as exp.# 1A but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would not be saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment#1A but now Back button is used for ending the AUT.

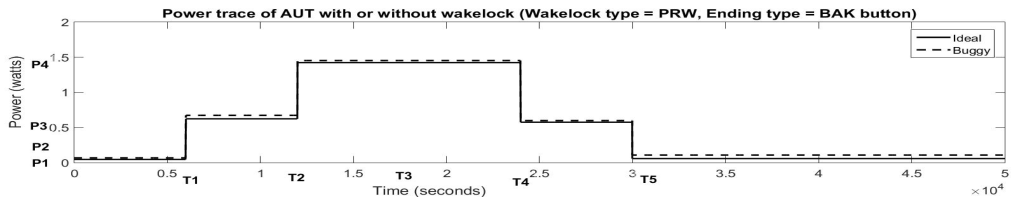


Figure 5.4: Power trace comparison of wakelock app (Experiment # 2A)

Result from App Tail ebug tool: Same as experiment number 1A.
Observations: Same as experiment#1A.
Conclusion: Same as experiment#1A. Using Back button for ending does not release Partial wakelock(s) acquired by the AUT. Service continue running in the background.
Experiment # 3A (Wakelock type : PRW, Ending type : Swipe-out)

Objective: Same as exp.# 1A but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

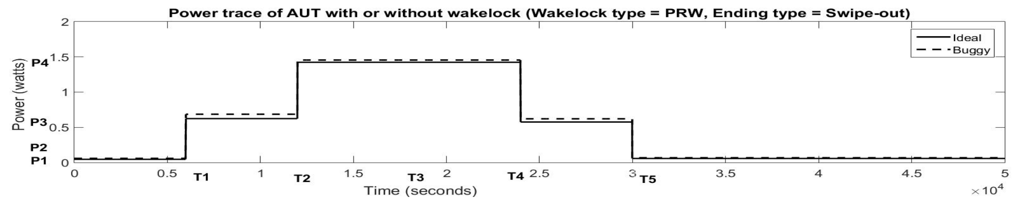


Figure 5.5: Power trace comparison of wakelock app (Experiment # 3A)

Observations: No more power consumption after ending the AUT
Conclusion: AUT had only one activity. Using swipe-out gesture for ending killed the only activity that is why process got killed as well Thats why all resources including wakelocks has been released.

Experiment # 4A (Wakelock type : PRW, Ending type : Force-stop)

Objective: Same as exp.#1A but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as exp.#1A However now Back button is used for ending the AUT.

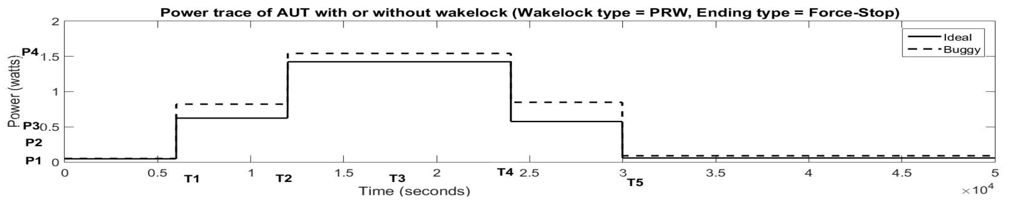


Figure 5.6: Power trace comparison of wakelock app (Experiment # 4A)

Result from App Tail ebug tool: No Wakelocks left.
Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity. Using force-stop for ending killed the process including activities, services etc. Thats why all resources including wakelocks has been released.

Experiment # 5A (Wakelock type : SDW, Ending type : Home button)

Objective: Same as exp.# 1A but Home button would be used for ending the AUT.
Hypothesis: : Current screen (activity) would be saved in the memory and then phone would displays Home screen. Acquired wakelocks would not be released.
Procedure: Same as exp.#1A However now Back button is used for ending the AUT.

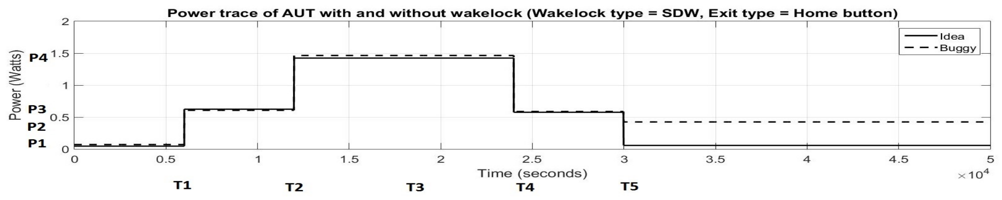


Figure 5.7: Power trace comparison of wakelock app (Experiment # 5A)

Result from App Tail ebug tool: Shows SDW wakelock still active.
Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1A. Using Home button for ending does not release Screen DIM wakelock(s) acquired by the AUT.

Experiment # 6A (Wakelock type : SDW, Ending type : Back button)

Objective: Same as experiment# 1A but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.

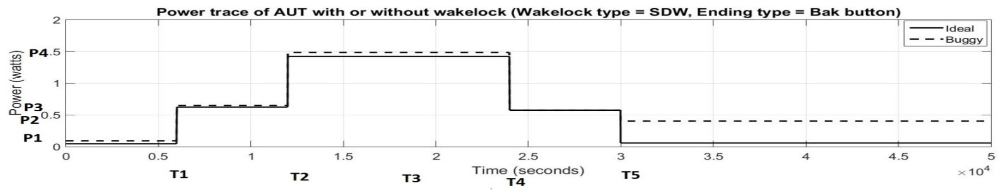


Figure 5.8: Power trace comparison of wakelock app (Experiment # 6A)

Result from App Tail ebug tool: Shows SDW wakelock still active.
Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1A. Using Home button for ending does not release Screen DIM wakelock(s) acquired by the AUT.

Experiment # 7A (Wakelock type : SDW, Ending type : Swipe-out)

Objective: Same as experiment# 1A but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

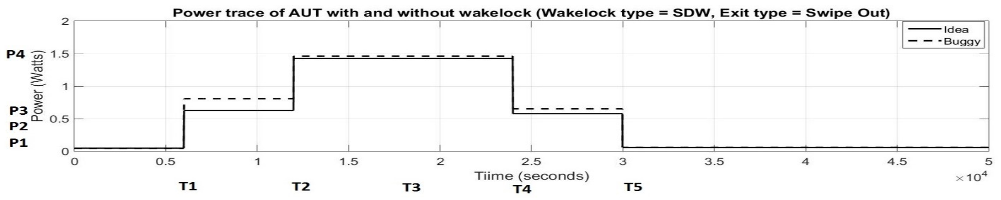


Figure 5.9: Power trace comparison of wakelock app (Experiment # 7A)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity. Using swipe-out gesture for ending killed the only activity that is why process got killed as well Thats why all resources including wakelocks has been released.

Experiment # 8A (Wakelock type : SDW, Ending type : Force-stop)

Objective: Same as experiment#1A but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment# 1A but now force-stop is used for ending the AUT.

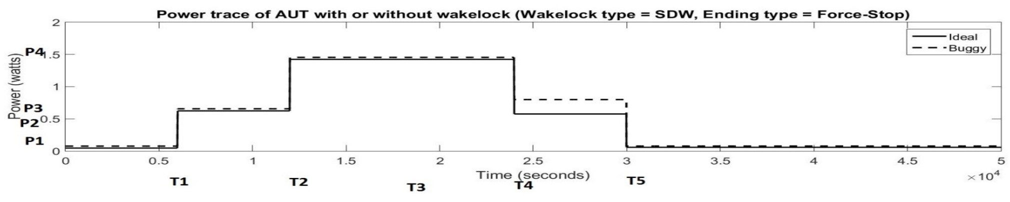


Figure 5.10: Power trace comparison of wakelock app (Experiment # 8A)

Result from App Tail ebug tool: No Wakelocks left.
Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity. Using force-stop for ending killed the process including activities, services etc. Thats why all resources including wakelocks has been released.

Experiment # 9A (Wakelock type : SBW, Ending type : Home button)

Objective: Same as experiment# 1A but Home button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be saved in the memory and then phone would displays Home screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Home button is used for ending the AUT.
Result from App Tail ebug tool: Shows SDW wakelock still active.

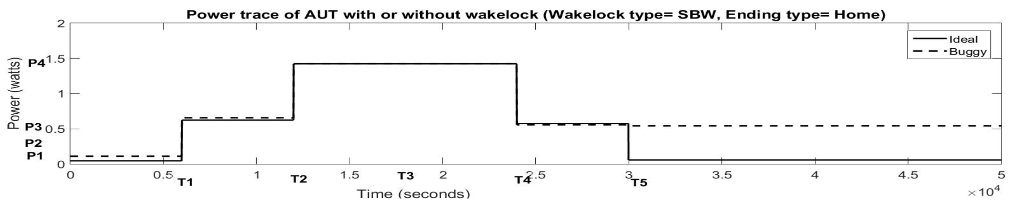


Figure 5.11: Power trace comparison of wakelock app (Experiment # 9A)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1A. Using Home button for ending does not release Screen DIM wakelock(s) acquired by the AUT.

Experiment # 10A (Wakelock type : SBW, Ending type : Back button)

Objective: Same as experiment# 1A but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.

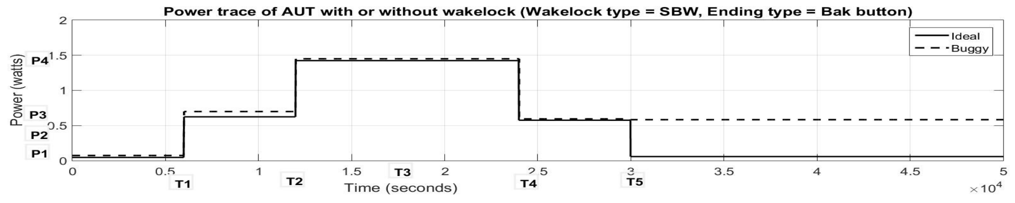


Figure 5.12: Power trace comparison of wakelock app (Experiment # 10A)

Result from App Tail ebug tool: Shows SDW wakelock still active.
Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1A. Using Home button for ending does not release Screen DIM wakelock(s) acquired by the AUT.

Experiment # 11A (Wakelock type : SBW, Ending type : Swipe-out)

Objective: Same as experiment# 1A but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

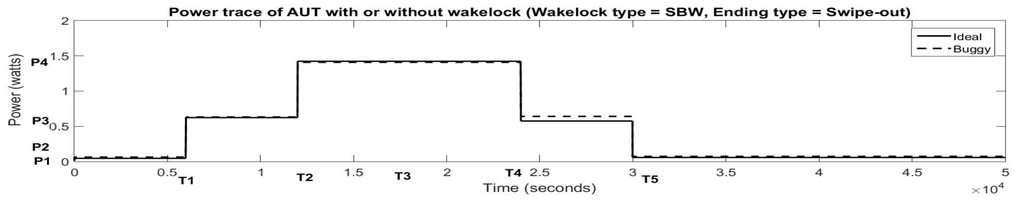


Figure 5.13: Power trace comparison of wakelock app (Experiment # 11A)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity. Using swipe-out gesture for ending killed the only activity that is why process got killed as well Thats why all resources including wakelocks has been released.

Experiment # 12A (Wakelock type : SBW, Ending type : Force-stop)

Objective: Same as experiment#1A but force-stop would be used for ending the AUT.
Hypothesis: : Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment# 1A but now force-stop is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

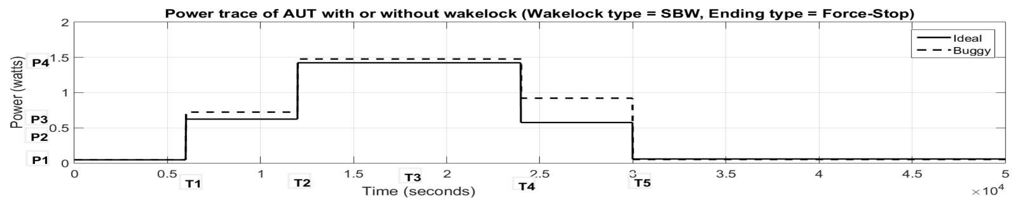


Figure 5.14: Power trace comparison of wakelock app (Experiment # 12A)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity. Using force-stop for ending killed the process including activities, services etc. Thats why all resources including wakelocks has been released.

Experiment # 13A (Wakelock type : FLW, Ending type : Home button)

Objective: Same as experiment# 1A but Home button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be saved in the memory and then phone would displays Home screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Home button is used for ending the AUT.
Result from App Tail ebug tool: Shows SDW wakelock still active.

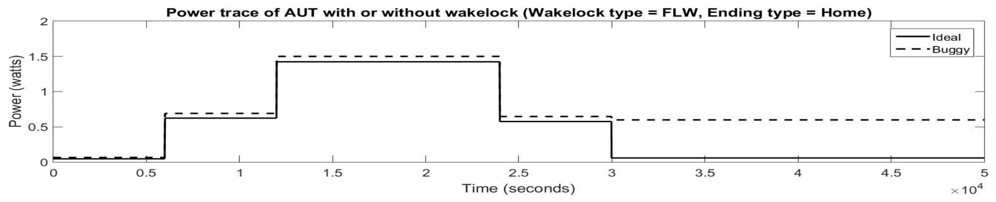


Figure 5.15: Power trace comparison of wakelock app (Experiment # 13A)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1A. Using Home button for ending does not release Full wakelock(s) acquired by the AUT.

Experiment # 14A (Wakelock type : FLW, Ending type : Back button)

Objective: Same as experiment# 1A but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows SDW wakelock still active.

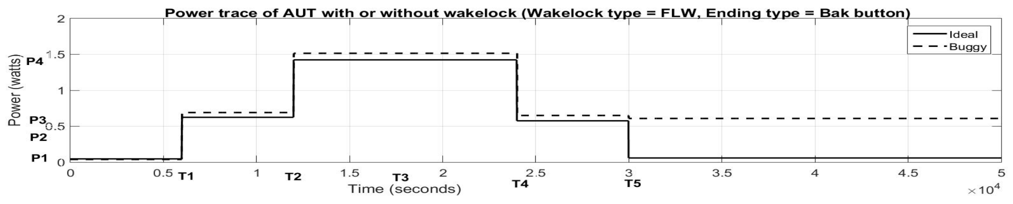


Figure 5.16: Power trace comparison of wakelock app (Experiment # 14A)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1A. Using Back button for ending does not release Full wakelock(s) acquired by the AUT.

Experiment # 15A (Wakelock type : FLW, Ending type : Swipe-out)

Objective: Same as experiment# 1A but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

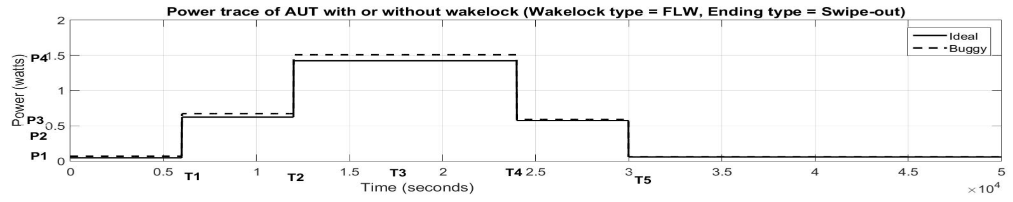


Figure 5.17: Power trace comparison of wakelock app (Experiment # 15A)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity. Using swipe-out gesture for ending killed the only activity that is why process got killed as well Thats why all resources including wakelocks has been released.

Experiment # 16A (Wakelock type : FLW, Ending type : Force-stop)

Objective: Same as experiment#1A but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment# 1A but now force-stop is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

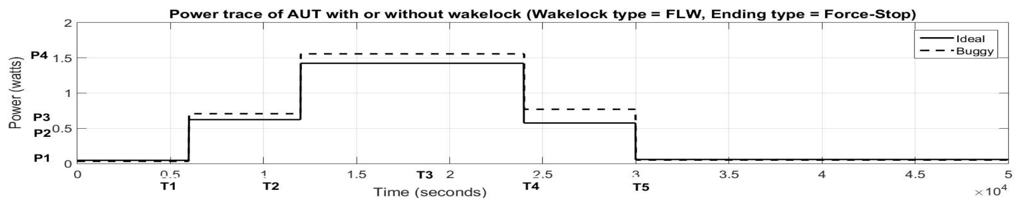


Figure 5.18: Power trace comparison of wakelock app (Experiment # 16A)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity. Using force-stop for ending killed the process including activities, services etc. Thats why all resources including wakelocks has been released.

Experiment # 17A (Wakelock type : No WKL, Ending type : Home button)

Objective: Same as exp.# 1A but Home button would be used for ending the AUT.
Hypothesis: Current screen (activity) would not be saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.Current screen (activity) would be saved in the memory and then phone would display Home screen.
Procedure: Same as experiment# 1A but now Home button is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

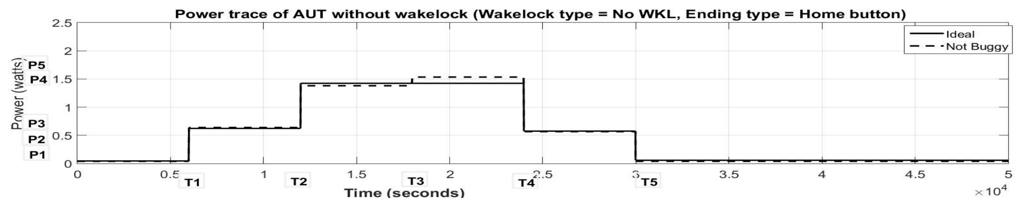


Figure 5.19: Power trace comparison of wakelock app (Experiment # 17A)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout.
Conclusion: Same as experiment # 1A. Using Home button for ending stops AUT.

Experiment # 18A (Wakelock type : No WKL, Ending type : Back button)

Objective: Same as experiment# 1A but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen.
Procedure: Same as experiment# 1A but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

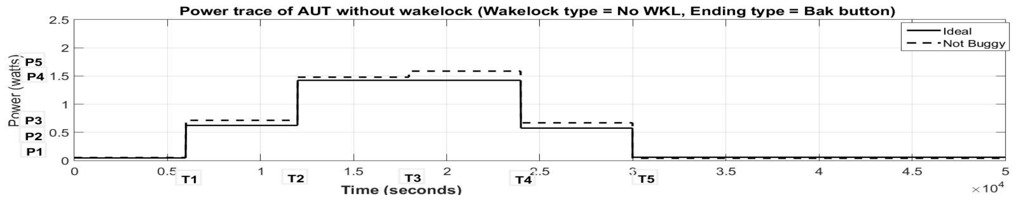


Figure 5.20: Power trace comparison of wakelock app (Experiment # 18A)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout.
Conclusion: Same as experiment # 1A. Using Back button for ending is stops AUT.

Experiment # 19A (Wakelock type : No WKL, Ending type : Swipe-out)

Objective: Same as experiment# 1A but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would not be saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as exp.# 1A but now swipe-out gesture is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

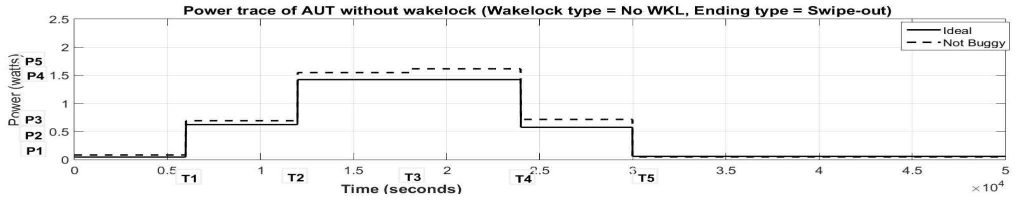


Figure 5.21: Power trace comparison of wakelock app (Experiment # 19A)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout.
Conclusion: Same as experiment # 1A. Using swipe-out for ending stops AUT.

Experiment # 20A (Wakelock type : No WKL, Ending type : Force-stop)

Objective: Same as experiment#1A but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed.
Procedure: Same as experiment# 1A but now force-stop is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

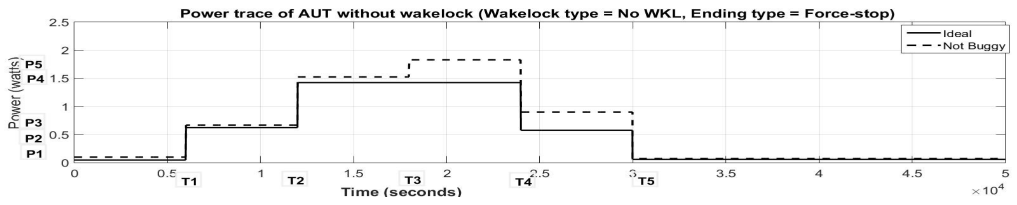


Figure 5.22: Power trace comparison of wakelock app (Experiment # 20A)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout.
Conclusion: Same as experiment # 1A. Using force-stop for ending stops AUT.

5.2 Test Cases for "Service"

Experiment # 1B (Wakelock type : PRW, Ending type :HOME button)
Objective: To monitor systems behaviour when a partial wakelock is not released before exiting due to specific user action using Home button. This wakelock is acquired by the AUT for a custom service,
Hypothesis: In this experiment, we acquire a partial wakelock for a custom service but not release it deliberately to simulate a real life environment. Then, we start and play with another app such as browser app for a while. Later, we come back to the wakelock demo app and close the app using Home button. As per default behaviour of Home button, current screen (activity) should be saved in the memory and then Home screen should be displayed. After display time out, previously started custom service will continue running in the background due to presence of wakelock. By default Android system does not take any action to release previously acquired wakelocks. To prove this assumption, we are doing following experiment.
Procedure: In order to check any deviation from ideal behaviour, we have run two test cases. In the Figure 5.23, the dotted line curve represents the power consumption of the app exhibiting buggy behaviour, while solid-line curve represent the normal behaviour. In beginning of the test, the smartphone is in the switched off state. We start the phone by pressing the power button. After successful boot up, the smartphone Home screen is completely visible. We do not touch the smartphones screen, and it remains in the same state for a while. At this stage, the device is in the idle state. Due to the aggressive sleeping policy implemented by the Android platform, the device goes into the sleeping mode. At this moment, we start to measure the power consumption of the device which corresponds to P1. We also click on pre-state button on app tail ebug tool on developers PC. Here, the device power consumption is 100 times less than the consumption in the On state; In order to run the test case, we press the power button of the smartphone to wake it up at time T1. The power consumption shots up from P1 to P3. Now, the display is visible, and other important components such as the CPU, display, keyboard and radio are also in full wake up state. At this stage, no app is running on the smartphone. We measured the power consumption P3 during time interval (T1-T2), which corresponds to the app pre-execution state. At time T2, we start the AUT by clicking on the app icon. The power consumption of the device shots up from P3 to P4. The wakelock demo app is loaded and it starts running with full functionality. Now, the app is in the execution state. We test different features of the application. At time T3 we press next screen button to the accesses WakelockAndriod2 screen. We measure the power consumption P4 during the time interval (T2-T3), which corresponds to the power consumption of the app during

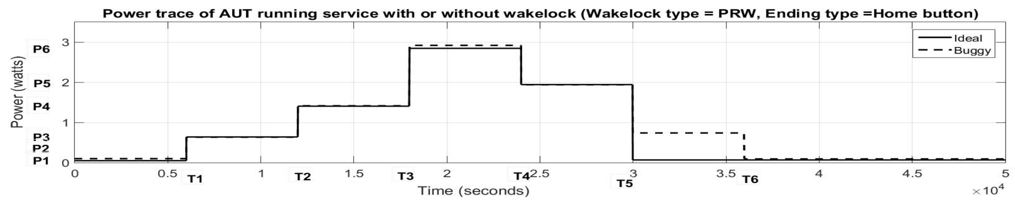
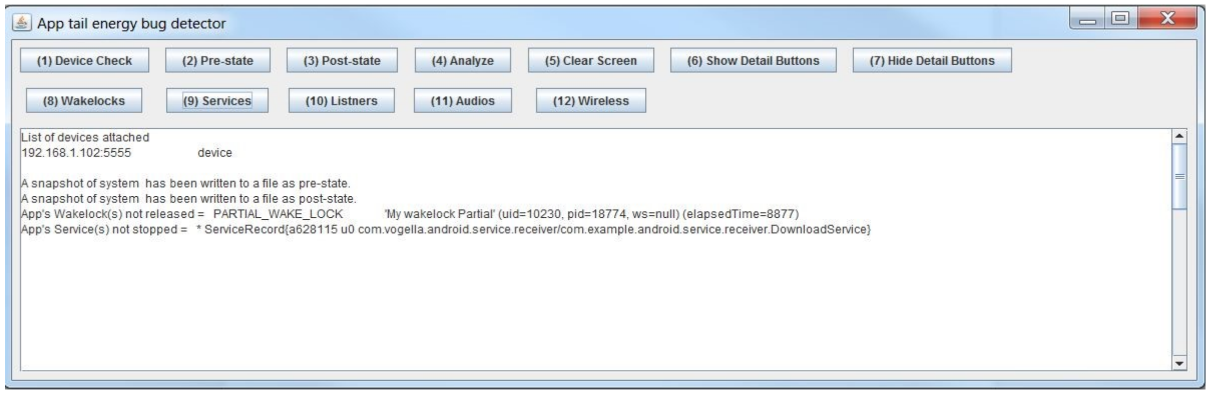


Figure 5.23: Power trace comparison of wakelock app (Experiment # 1B)

execution. At time T3, we also pressed start download button to start a service and then browser button to start a browser. Due to file download service, power consumption shots up from P4 to P6. After a while, we come back to same screen of wakelock demo app. At time T4, the app is stopped by pressing the Home button from the WakelockAndriod2 screen, as shown in Figure 1.3. Power consumption drop from P6 to P5 due to different screen resolutions of home screen. We do not touch the screen. At time T5, system goes to sleeping mode after screen time out. In our case we setup screen time out two minutes (120 seconds). At this point in case of a correct app power drops from P5 to P1 where as in case of buggy app it drops from P5 to P4 and stays there till file download is complete. At time T6. Power consumption drops from P4 to P2 and stay there due to partial wakelock. For simplicity, all steps mentioned above are summarized in Table 5.25 Starting from T5, the power consumption of the app in the absence of wakelock is completely differs from that of the presence of wakelock. In the absence of wakelock, the app power consumption has dropped from P5 to P1. Once the foreground screen is moved to the background, the AUT and file download should be stopped and wakelock should be released. In contrast, in the presence of the wakelock, the app power consumption did not drop to P1 level and stayed on the P2 level. After the T4, the app is considered to be in the post-execution state. We measure the power consumption during the time interval (T4-T5) and call it the post-execution power consumption of the app. After stopping, we do not touch the device at all. After display time out at T5, the power consumption of the app without a wakelock drops from P5 to P1, whereas with wakelock, it drops from P5 to P4 and then P4 to P2 after file download is completed.

Result from App Tail ebug tool: It shows a custom service is still running with the help of a partial wakelock even after display time out.

Observations: As shown in Figure 5.23, the graph shows that in normal case without a service, power consumption is always settles down to the pre-execution power state level after stopping the AUT. The device goes to suspend mode after a while when wakelock is not present, whereas in case of a buggy behaviour, wakelock does not allow the device to go into the sleep mode. As shown in Figure 5.24, the results shows from app tail ebug tool



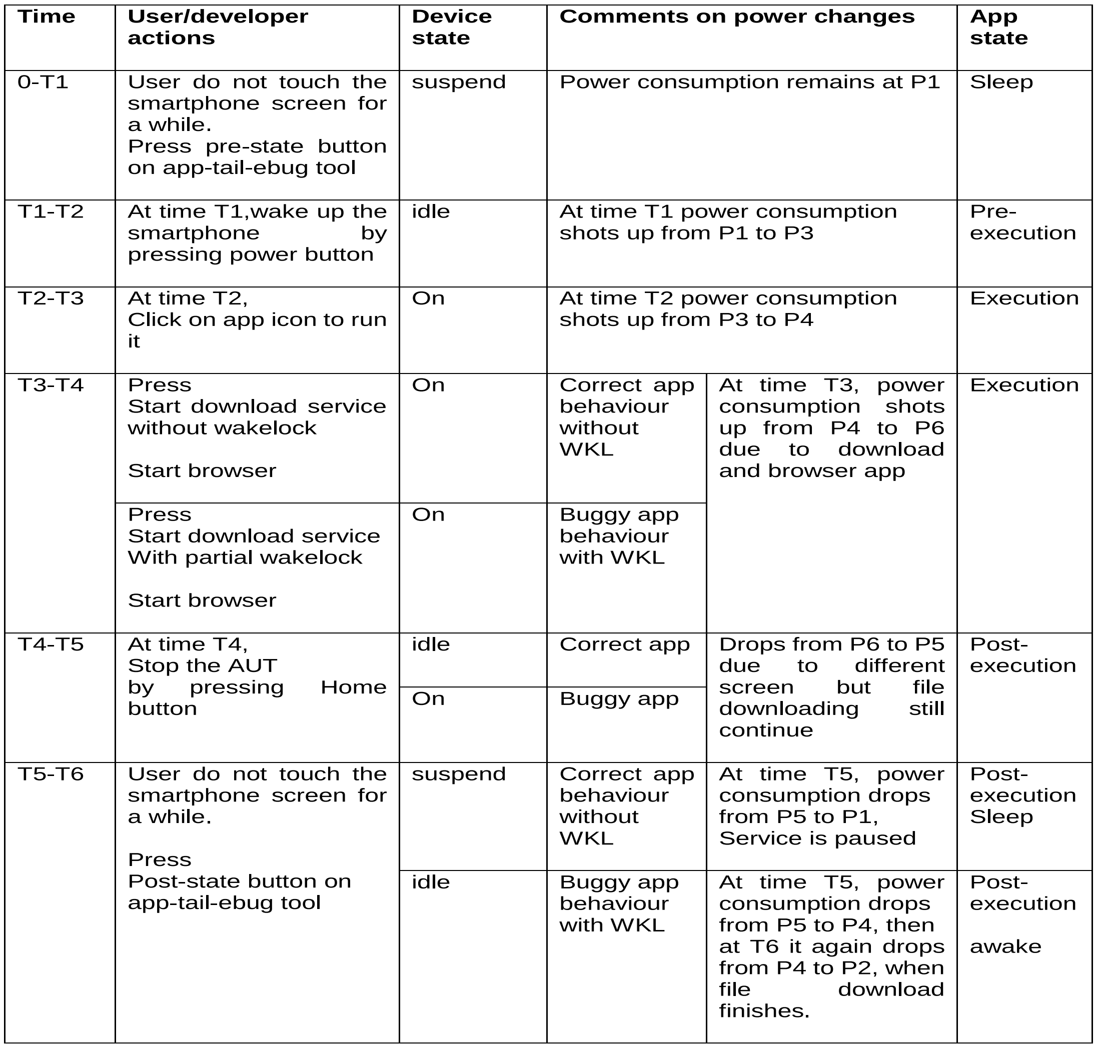
```
List of devices attached
192.168.1.102:5555              device

A snapshot of system  has been written to a file as pre-state.
A snapshot of system  has been written to a file as post-state.
App's Wakelock(s) not released =   PARTIAL_WAKE_LOCK          'My wakelock Partial' (uid=10230, pid=18774, ws=null) (elapsedTime=8877)
App's Service(s) not stopped =   * ServiceRecord{a628115 u0 com.vogella.android.service.receiver/com.example.android.service.receiver.DownloadService}
```

Figure 5.24: Result from application tail energy bugs tool (Experiment # 1B)

also supports our theory mentioned above. Our tool successfully detects root cause of the problem. A partial wakelock initiated to run download service, which is causing the loss of battery power in the buggy app.

Conclusion: We conducted an experiment to show the presence of the app tail energy bug when a partial wakelocks is not handled properly. We simulated the real environment by using a wakelock demo app. We acquired a partial wakelock from AUT but did not release it deliberately to check its affects after closing the app. While playing with AUT, we started a download service and then switched to a browser app and then come back to AUT after a short while. We used Home button to close the AUT. After a display timeout of 120 seconds, screen turned to dim and then completely dark. Partial wakelock keeps the CPU awake but nothing to do with display. The download service did not stop even after display time out. The continuous power consumption after display is off shows the presence of partial wakelock which triggered at right time. Though it helped to run download service in the background. The file has been downloaded completely. Using Home button for exiting do not release wakelock(s) acquired by the AUT. In the other experiment, we did not acquire any wakelock, the running service pauses when the device went into sleep. Later the same service resume when the device awake. In the first experiment full file has been download but in another experiment system killed the background service without downloading complete file.



Time	User/developer actions	Device state	Comments on power changes		App state
0-T1	User do not touch the smartphone screen for a while. Press pre-state button on app-tail-ebug tool	suspend	Power consumption remains at P1		Sleep
T1-T2	At time T1,wake up the smartphone by pressing power button	idle	At time T1 power consumption shots up from P1 to P3		Pre-execution
T2-T3	At time T2, Click on app icon to run it	On	At time T2 power consumption shots up from P3 to P4		Execution
T3-T4	Press Start download service without wakelock Start browser	On	Correct app behaviour without WKL	At time T3, power consumption shots up from P4 to P6 due to download and browser app	Execution
	Press Start download service With partial wakelock Start browser	On	Buggy app behaviour with WKL		
T4-T5	At time T4, Stop the AUT by pressing Home button	idle	Correct app	Drops from P6 to P5 due to different screen but file downloading still continue	Post-execution
		On	Buggy app		
T5-T6	User do not touch the smartphone screen for a while. Press Post-state button on app-tail-ebug tool	suspend	Correct app behaviour without WKL	At time T5, power consumption drops from P5 to P1, Service is paused	Post-execution Sleep
		idle	Buggy app behaviour with WKL	At time T5, power consumption drops from P5 to P4, then at T6 it again drops from P4 to P2, when file download finishes.	Post-execution awake

Figure 5.25: Summary of the main states of the AUT during test case execution.

Experiment # 2B (Wakelock type : PRW, Ending type : Back button)
Objective: Same as experiment# 1B but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would not be saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Same as experiment#1B.

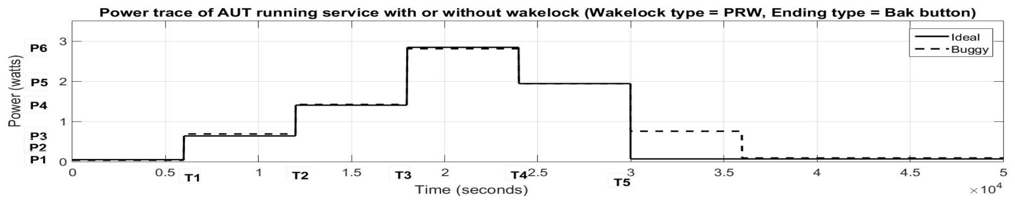


Figure 5.26: Power trace comparison of wakelock app (Experiment # 2B)

Observations: Same as experiment#1B.
Conclusion: Same as experiment # 1B. Using Back button for ending does not release Partial wakelock(s) acquired by the AUT. Service continue in the background.

Experiment # 3B (Wakelock type : PRW, Ending type : Swipe-out)
Objective: Same as experiment#1B but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: PRW still active

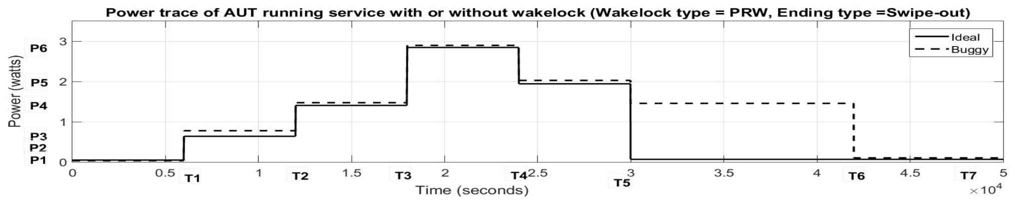


Figure 5.27: Power trace comparison of wakelock app (Experiment # 3B)

Observations: power consumption continue after ending the AUT
Conclusion: AUT had one activity and one service. Using swipe-out gesture for ending killed the activity and service but service is restarted and running continuously till it finished intended task. Power consumption continued due to PRW.

Experiment # 4B (Wakelock type : PRW, Ending type : Force-stop)

Objective: Same as experiment#1B but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment# 1B but now force-stop is used for ending the AUT.

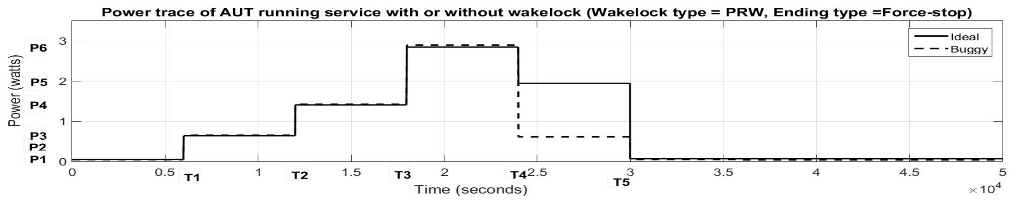


Figure 5.28: Power trace comparison of wakelock app (Experiment # 4B)

Result from App Tail ebug tool: No Wakelocks left.
Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity and a service. Using force-stop for ending killed the process including all activities and services etc. Thats why all resources including wakelocks has been released.

Experiment # 5B (Wakelock type : SDW, Ending type : Home button)
Objective: Same as exp.# 1B but Home button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be saved in the memory and then phone would displays Home screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Home button is used for ending the AUT.
Result from App Tail ebug tool: Shows SDW wakelock still active.

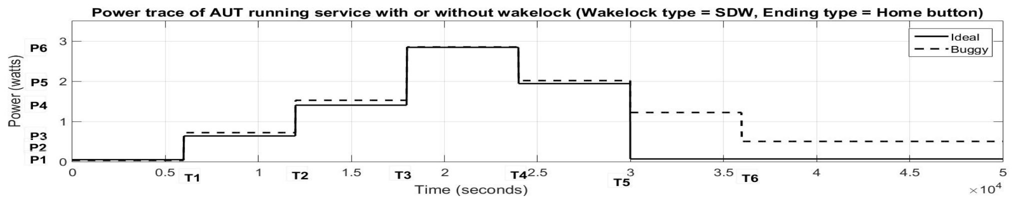


Figure 5.29: Power trace comparison of wakelock app (Experiment # 5B)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1B. Using Home button for ending does not release Screen DIM wakelock(s) acquired by the AUT.

Experiment # 6B (Wakelock type : SDW, Ending type : Back button)
Objective: Same as experiment# 1B but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.

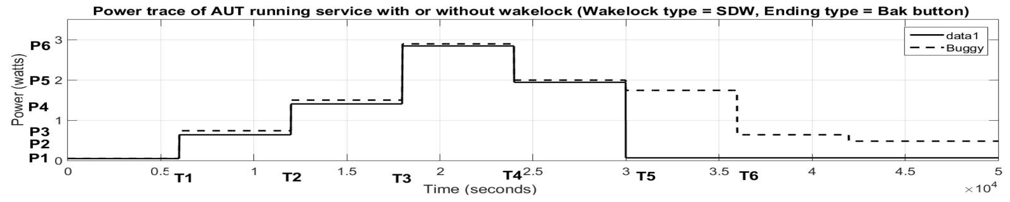


Figure 5.30: Power trace comparison of wakelock app (Experiment # 6B)

Result from App Tail ebug tool: Shows SDW wakelock still active.
Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1B. Using Home button for ending does not release Screen DIM wakelock(s) acquired by the AUT.

Experiment # 7B (Wakelock type : SDW, Ending type : Swipe-out)
Objective: Same as experiment# 1B but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows SDW still active

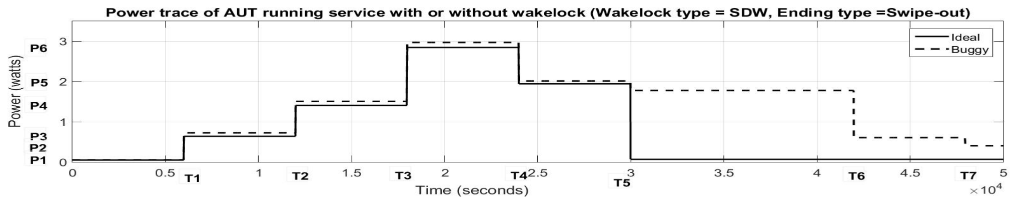


Figure 5.31: Power trace comparison of wakelock app (Experiment # 7B)

Observations: power consumption continue after ending the AUT
Conclusion: AUT had one activity and one service. Using swipe-out gesture for ending killed the activity and service but service is restarted and running continuously till it finished intended task. Power consumption continued due to SDW.

Experiment # 8B (Wakelock type : SDW, Ending type : Force-stop)
Objective: Same as experiment#1B but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment# 1B but now force-stop is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

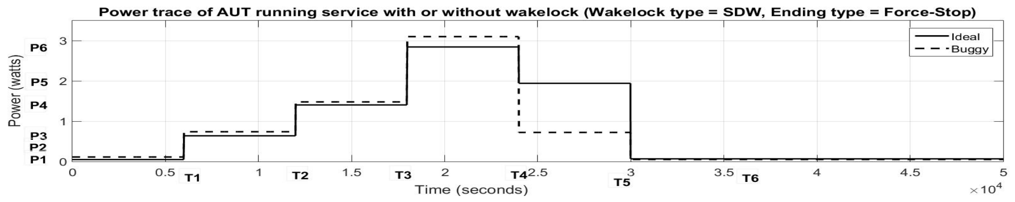


Figure 5.32: Power trace comparison of wakelock app (Experiment # 8B)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had one activity and one service. Using force-stop for ending killed the process including activities, services etc. Thats why all resources including wakelocks has been released.

Experiment # 9B (Wakelock type : PRW, Ending type : Home button)
Objective: Same as experiment# 1B but Home button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be saved in the memory and then phone would displays Home screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Home button is used for ending the AUT.
Result from App Tail ebug tool: Shows SBW wakelock still active.

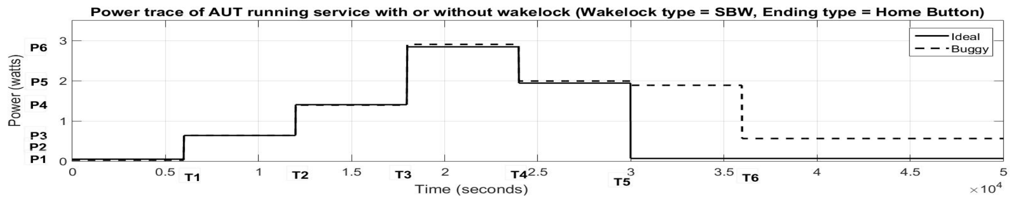


Figure 5.33: Power trace comparison of wakelock app (Experiment # 9B)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1B. Using Home button for ending does not release Screen Bright wakelock(s) acquired by the AUT.

Experiment # 10B (Wakelock type : SBW, Ending type : Back button)
Objective: Same as experiment# 1B but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows SDW wakelock still active.

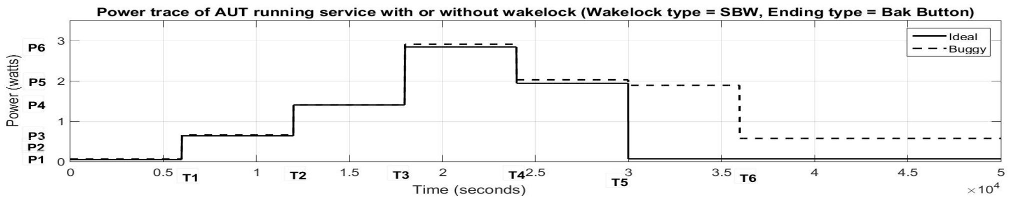


Figure 5.34: Power trace comparison of wakelock app (Experiment # 10B)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1B. Using Home button for ending does not release Screen Bright wakelock(s) acquired by the AUT.

Experiment # 11B (Wakelock type : SBW, Ending type : Swipe-out)
Objective: Same as experiment# 1B but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would not be saved in the memory and then phone would dispCurrent screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows SBW still active

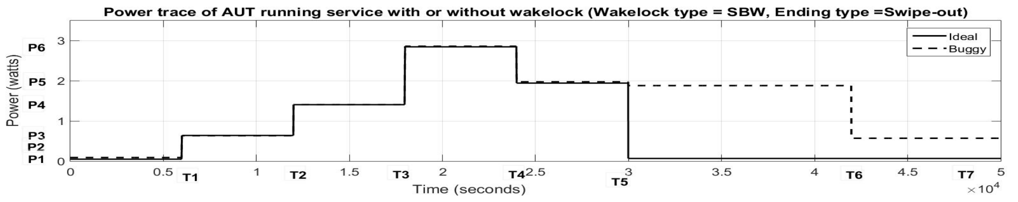


Figure 5.35: Power trace comparison of wakelock app (Experiment # 11B)

Observations: power consumption continue after ending the AUT.
Conclusion: AUT had only one activity and one service. Using swipe-out gesture for ending killed the activity and service but service re-started and continue running till it finished intended task. Thats why all resources including wakelocks has not been released.

Experiment # 12B (Wakelock type : SBW, Ending type : Force-stop)
Objective: Same as experiment#1B but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment# 1B but now force-stop is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

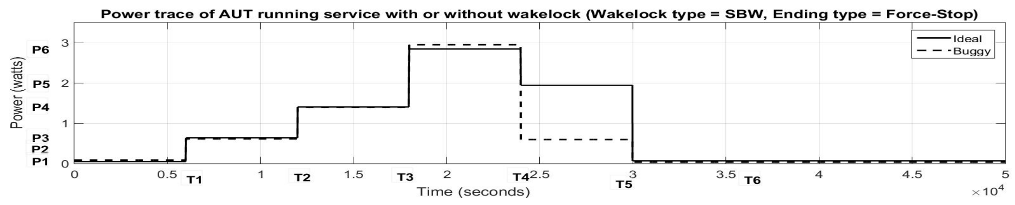


Figure 5.36: Power trace comparison of wakelock app (Experiment # 12B)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity and one service. Using force-stop for ending killed the process including activities, services etc. Thats why all resources including wakelocks has been released.

Experiment # 13B (Wakelock type : FLW, Ending type : Home button)
Objective: Same as experiment# 1B but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows FLW wakelock still active.

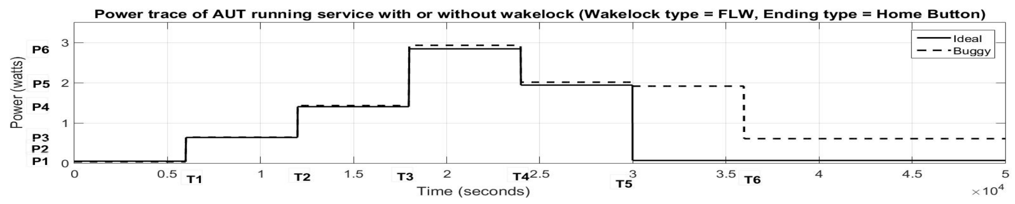


Figure 5.37: Power trace comparison of wakelock app (Experiment # 13B)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1B. Using Home button for ending does not release FLW wakelock(s) acquired by the AUT.

Experiment # 14B (Wakelock type : FLW, Ending type : Back button)

Objective: Same as exp.# 1B but Back button would be used for ending the AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows FLW wakelock still active.

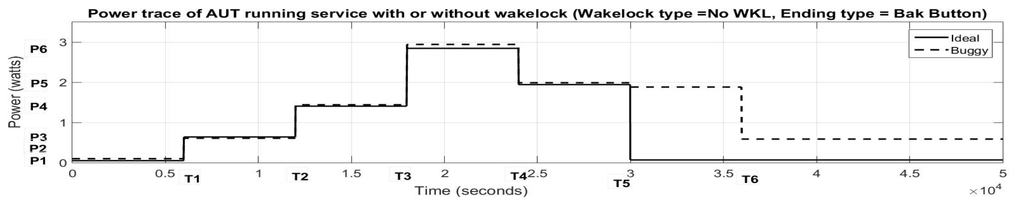


Figure 5.38: Power trace comparison of wakelock app (Experiment # 14B)

Observations: Power consumption is continued even after ending the AUT.
Conclusion: Same as experiment # 1B. Using Back button for ending does not release Full wakelock(s) acquired by the AUT.

Experiment # 15B (Wakelock type : FLW, Ending type : Swipe-out)
Objective: Same as exp.# 1B but swipe-out gesture would be used for ending the AUT.
Hypothesis: Current screen (activity) would be killed and then process. Acquired wakelocks would not be released.
Procedure: Same as experiment# 1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows FLW still active#1B.

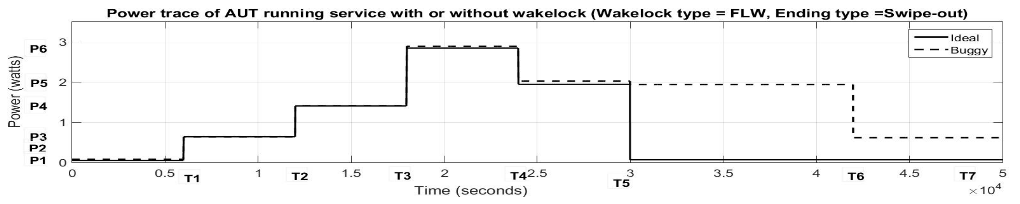


Figure 5.39: Power trace comparison of wakelock app (Experiment # 15B)

Observations: power consumption continua after ending the AUT
Conclusion: AUT had only one activity and one service. Using swipe-out gesture for ending killed the only activity and service but service is restarted and continue running till it finished intended task. Thats why all resources including wakelocks has not been released.

Experiment # 16B (Wakelock type : FLW, Ending type : Force-stop)
Objective: Same as experiment#1B but force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment# 1B but now force-stop is used for ending the AUT.
Result from App Tail ebug tool: No Wakelocks left.

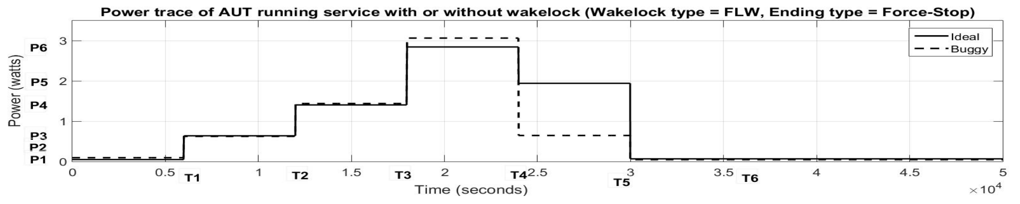


Figure 5.40: Power trace comparison of wakelock app (Experiment # 16B)

Observations: No more power consumption after ending the AUT.
Conclusion: AUT had only one activity and one service. Using force-stop for ending killed the process including activities, services etc. Thats why all resources including wakelocks has been released.

Experiment # 17B (Wakelock type : No WKL, Ending type : Home button)
Objective: Same as experiment#1B but Home button would be used for ending AUT.
Hypothesis: Current screen (activity) would be saved in the memory and then phone would displays Home screen.
Procedure: Same as experiment# 1B but Home button is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

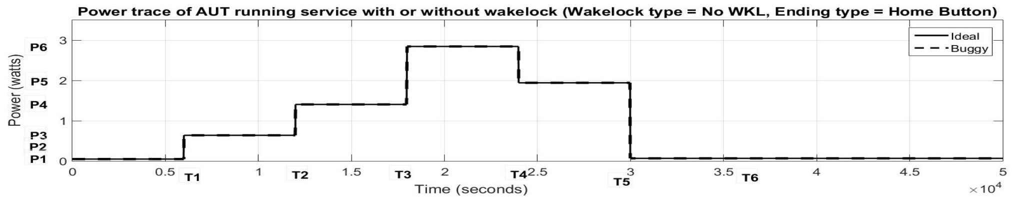


Figure 5.41: Power trace comparison of wakelock app (Experiment # 17B)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout. A background service stops temporarily when device goes to sleep.
Conclusion: Using Home button for ending does not help service to run continuously in the background without a wakelock.

Experiment # 18B (Wakelock type : No WKL, Ending type : Back button)
Objective: Same as experiment#1B but Back button would be used for ending AUT.
Hypothesis: Current screen (activity) would be not saved in the memory and then phone would displays previous screen.
Procedure: Same as experiment#1B but now Back button is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

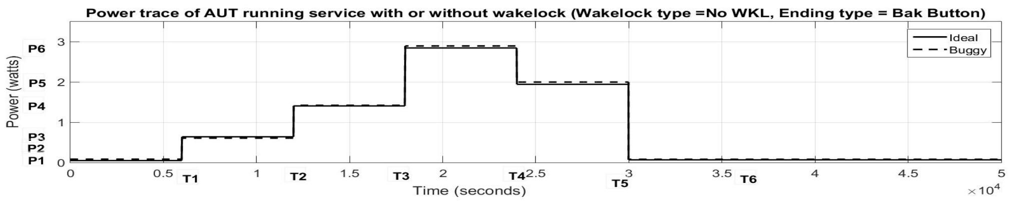


Figure 5.42: Power trace comparison of wakelock app (Experiment # 18B)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout. A background service stops temporarily when device goes to sleep.
Conclusion: Using Back button for ending does not help service to run continuously in the background without a wakelock.

Experiment # 19B (Wakelock type : No WKL, Ending type : Swipe-out)
Objective: Same as experiment#1B, swipe-out gesture would be used for ending AUT.
Hypothesis: Current screen (activity) would be killed and then process.
Procedure: Same as experiment#1B but now swipe-out is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

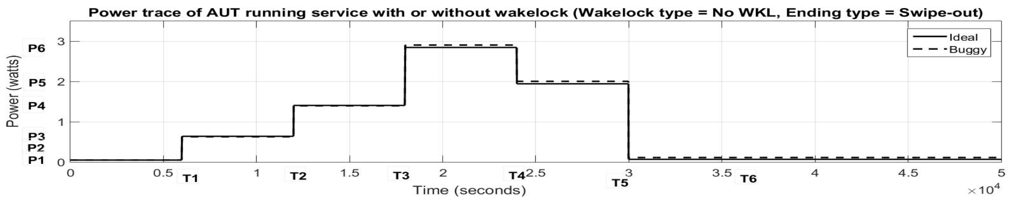


Figure 5.43: Power trace comparison of wakelock app (Experiment # 19B)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout. A background service stops temporarily when device goes to sleep.
Conclusion: Using swipe-out for ending does not help service to run continuously in the background without a wakelock.

Experiment # 20B (Wakelock type : No WKL, Ending type : Force-stop)
Objective: Same as experiment#1B, force-stop would be used for ending the AUT.
Hypothesis: Current screen (activity) including process would be killed. Acquired wakelocks would also be released.
Procedure: Same as experiment#1B but now force-stop is used for ending the AUT.
Result from App Tail ebug tool: Shows no wakelock still active.

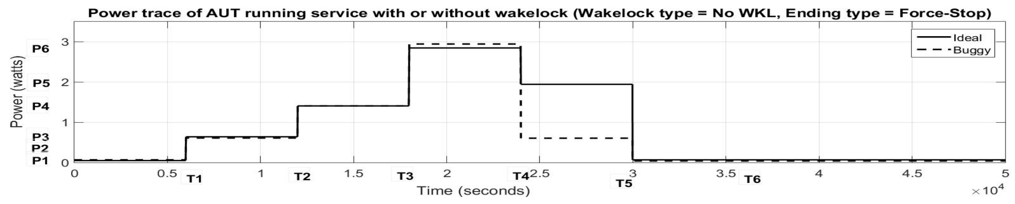


Figure 5.44: Power trace comparison of wakelock app (Experiment # 20B)

Observations: poststate power consumption settles down to pre-state. Device goes to sleep after display timeout. A background service stops temporarily when device goes to sleep.
Conclusion: Using force-stop for ending does not help service to run continuously in the background without a wakelock.

Experiment # 21B (Wakelock type : PRW released, Ending type : Home button)
Objective: Same as experiment#1B but Back button would be used for ending AUT.
Hypothesis: Current screen (activity) would be saved in the memory and then phone would displays Home screen. Acquired wakelocks would be released properly.
Procedure: Same as experiment#1B, Home button is used for ending the AUT.
Result from App Tail ebug tool: No Wakelock left.

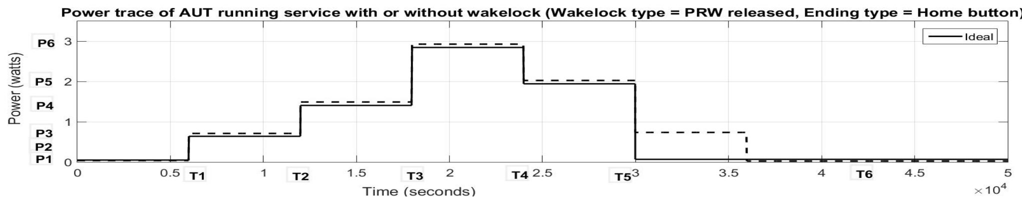


Figure 5.45: Power trace comparison of wakelock app (Experiment # 21B)

Observations: Same as experiment#1B.
Conclusion: Same as experiment#1B. Using swipe-out gesture for ending does not release Partial wakelock(s) acquired by the AUT. Service continue in the background till it finishes intended task. Wakelock has been released.

Experiment # 22B (Wakelock type : PRW released, Ending type : swipe-out)
Objective: Same as experiment#1B, swipe-out gesture would be used for ending AUT.
Hypothesis: Current screen (activity) would not be saved in the memory and then phone would displays previous screen. Acquired wakelocks would be released properly.
Procedure: Same as experiment#1B, swipe-out gesture is used for ending the AUT.
Result from App Tail ebug tool: No Wakelock left

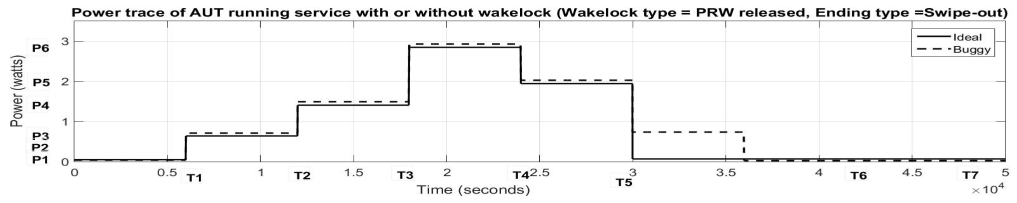
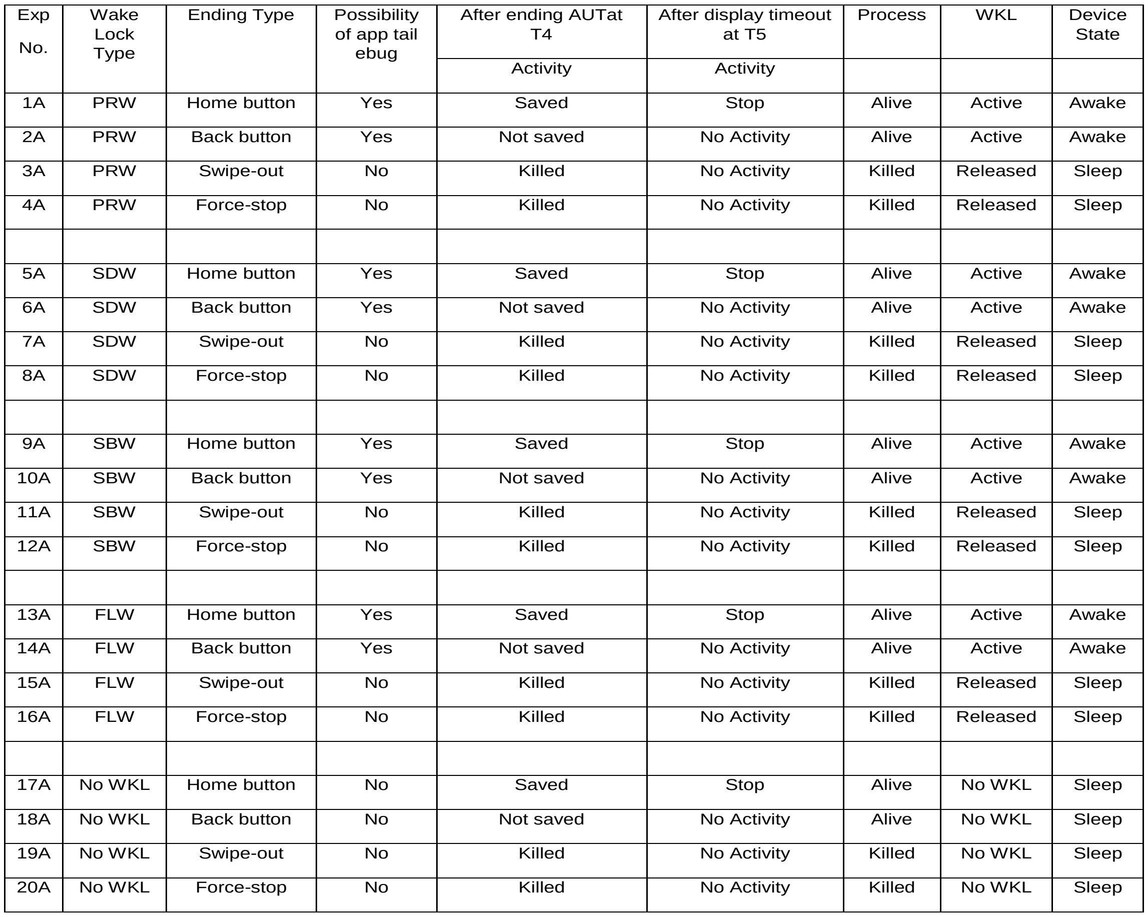


Figure 5.46: Power trace comparison of wakelock app (Experiment # 22B)

Observations: No power consumption after finishing intended task
Conclusion: Same as experiment#1B. Using swipe-out gesture for ending does not release Partial wakelock(s) acquired by the AUT. Service continue in the background till it finishes intended task. Wakelock has been released.

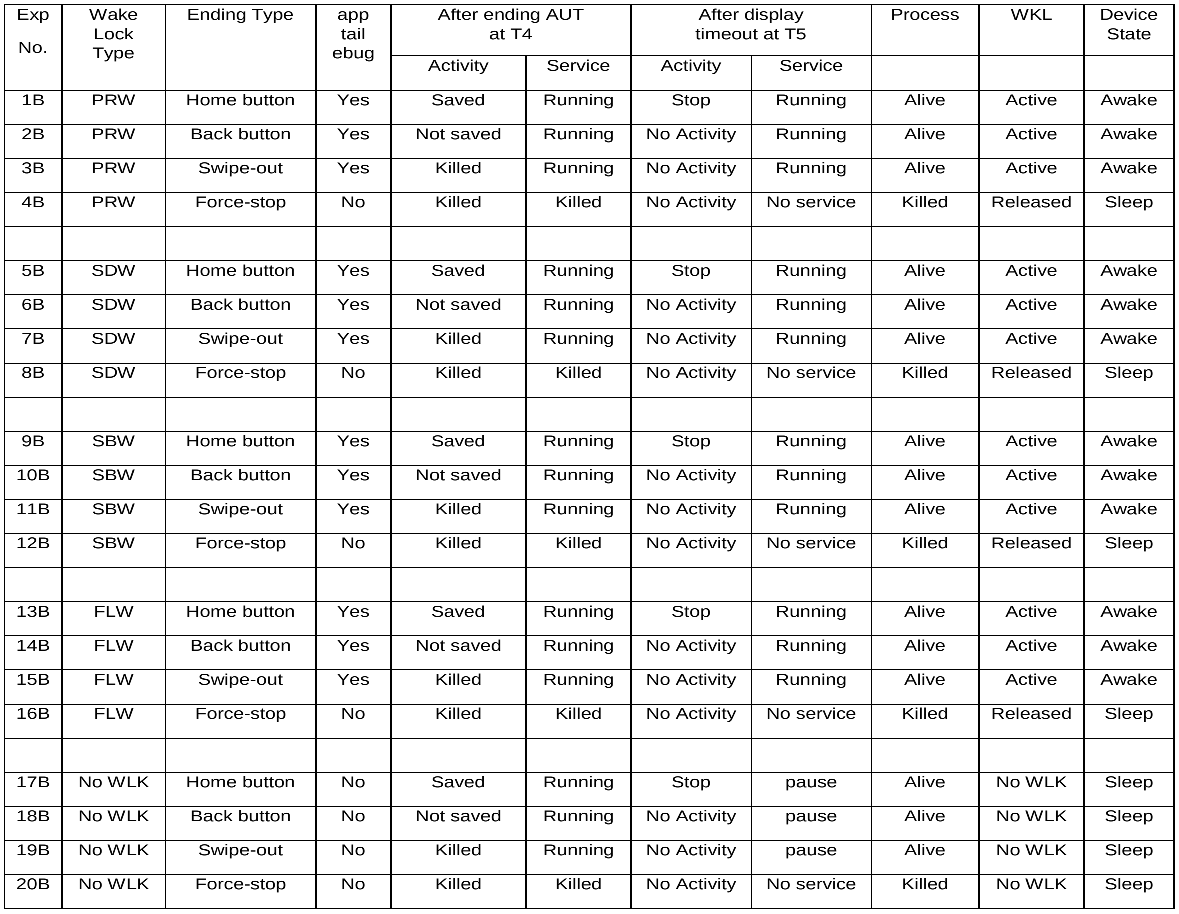
Exception: If you hold a partial wakelock, the CPU will continue to run, regardless of any display time out or the state of the screen and even after the user presses the power button. In all other wakelock types, the CPU will run, however the user can still put the device to sleep using the power button.

The table 5.49 summarizes results from all test cases. It can be observed that Activity never runs in the background with or without wakelocks that is why there is no chance of energy bug due to activities. In case of test cases AHW2 and ABW2, acquired wakelock must be released otherwise that can keep the device awake and consume battery. Another component of android app is service which runs in background without any UI. The only user action which kill the process including all activities, services, and release all acquired wakelocks. That is the reason except Force-stop, all test cases shows possibility of ebug. When device goes to sleep, background running services also get paused and resume whenever device re-awake. On the hand in the presence of any wakelock type, service can continue running in the background till it finishes intended task. All acquired wakelock has to be released once the the task is completed.



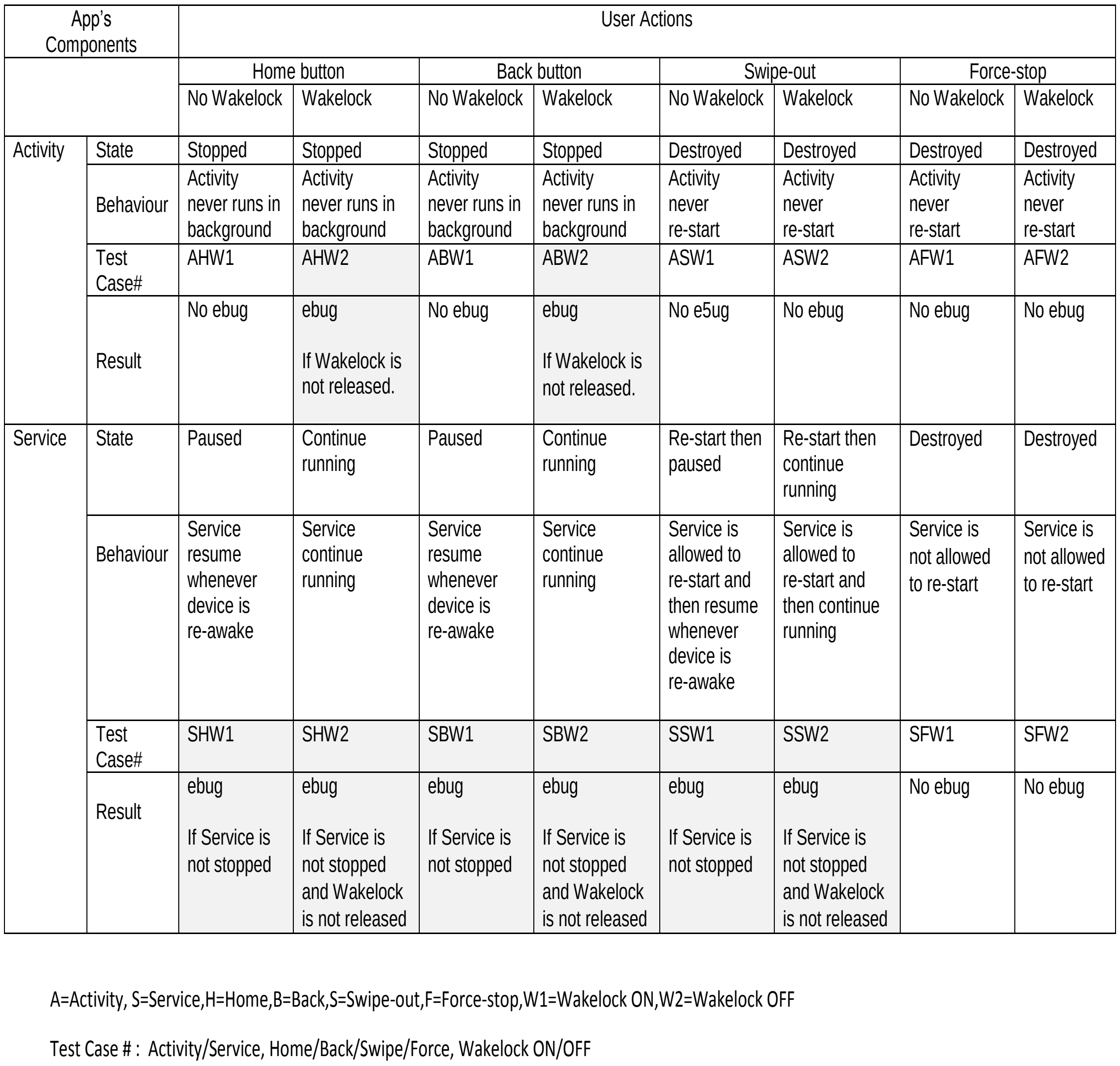
Exp No.	Wake Lock Type	Ending Type	Possibility of app tail ebug	After ending AUTat T4	After display timeout at T5	Process	WKL	Device State
				Activity	Activity			
1A	PRW	Home button	Yes	Saved	Stop	Alive	Active	Awake
2A	PRW	Back button	Yes	Not saved	No Activity	Alive	Active	Awake
3A	PRW	Swipe-out	No	Killed	No Activity	Killed	Released	Sleep
4A	PRW	Force-stop	No	Killed	No Activity	Killed	Released	Sleep
5A	SDW	Home button	Yes	Saved	Stop	Alive	Active	Awake
6A	SDW	Back button	Yes	Not saved	No Activity	Alive	Active	Awake
7A	SDW	Swipe-out	No	Killed	No Activity	Killed	Released	Sleep
8A	SDW	Force-stop	No	Killed	No Activity	Killed	Released	Sleep
9A	SBW	Home button	Yes	Saved	Stop	Alive	Active	Awake
10A	SBW	Back button	Yes	Not saved	No Activity	Alive	Active	Awake
11A	SBW	Swipe-out	No	Killed	No Activity	Killed	Released	Sleep
12A	SBW	Force-stop	No	Killed	No Activity	Killed	Released	Sleep
13A	FLW	Home button	Yes	Saved	Stop	Alive	Active	Awake
14A	FLW	Back button	Yes	Not saved	No Activity	Alive	Active	Awake
15A	FLW	Swipe-out	No	Killed	No Activity	Killed	Released	Sleep
16A	FLW	Force-stop	No	Killed	No Activity	Killed	Released	Sleep
17A	No WKL	Home button	No	Saved	Stop	Alive	No WKL	Sleep
18A	No WKL	Back button	No	Not saved	No Activity	Alive	No WKL	Sleep
19A	No WKL	Swipe-out	No	Killed	No Activity	Killed	No WKL	Sleep
20A	No WKL	Force-stop	No	Killed	No Activity	Killed	No WKL	Sleep

Figure 5.47: Summary of results from all test cases related to "Activity"



Exp No.	Wake Lock Type	Ending Type	app tail ebug	After ending AUT at T4		After display timeout at T5		Process	WKL	Device State
				Activity	Service	Activity	Service			
1B	PRW	Home button	Yes	Saved	Running	Stop	Running	Alive	Active	Awake
2B	PRW	Back button	Yes	Not saved	Running	No Activity	Running	Alive	Active	Awake
3B	PRW	Swipe-out	Yes	Killed	Running	No Activity	Running	Alive	Active	Awake
4B	PRW	Force-stop	No	Killed	Killed	No Activity	No service	Killed	Released	Sleep
5B	SDW	Home button	Yes	Saved	Running	Stop	Running	Alive	Active	Awake
6B	SDW	Back button	Yes	Not saved	Running	No Activity	Running	Alive	Active	Awake
7B	SDW	Swipe-out	Yes	Killed	Running	No Activity	Running	Alive	Active	Awake
8B	SDW	Force-stop	No	Killed	Killed	No Activity	No service	Killed	Released	Sleep
9B	SBW	Home button	Yes	Saved	Running	Stop	Running	Alive	Active	Awake
10B	SBW	Back button	Yes	Not saved	Running	No Activity	Running	Alive	Active	Awake
11B	SBW	Swipe-out	Yes	Killed	Running	No Activity	Running	Alive	Active	Awake
12B	SBW	Force-stop	No	Killed	Killed	No Activity	No service	Killed	Released	Sleep
13B	FLW	Home button	Yes	Saved	Running	Stop	Running	Alive	Active	Awake
14B	FLW	Back button	Yes	Not saved	Running	No Activity	Running	Alive	Active	Awake
15B	FLW	Swipe-out	Yes	Killed	Running	No Activity	Running	Alive	Active	Awake
16B	FLW	Force-stop	No	Killed	Killed	No Activity	No service	Killed	Released	Sleep
17B	No WLK	Home button	No	Saved	Running	Stop	pause	Alive	No WLK	Sleep
18B	No WLK	Back button	No	Not saved	Running	No Activity	pause	Alive	No WLK	Sleep
19B	No WLK	Swipe-out	No	Killed	Running	No Activity	pause	Alive	No WLK	Sleep
20B	No WLK	Force-stop	No	Killed	Killed	No Activity	No service	Killed	No WLK	Sleep

Figure 5.48: Summary of results from all test cases related to ”Service”



App's Components		User Actions							
		Home button		Back button		Swipe-out		Force-stop	
		No Wakelock	Wakelock	No Wakelock	Wakelock	No Wakelock	Wakelock	No Wakelock	Wakelock
Activity	State	Stopped	Stopped	Stopped	Stopped	Destroyed	Destroyed	Destroyed	Destroyed
	Behaviour	Activity never runs in background	Activity never runs in background	Activity never runs in background	Activity never runs in background	Activity never re-start	Activity never re-start	Activity never re-start	Activity never re-start
	Test Case#	AHW1	AHW2	ABW1	ABW2	ASW1	ASW2	AFW1	AFW2
	Result	No ebug	ebug If Wakelock is not released.	No ebug	ebug If Wakelock is not released.	No e5ug	No ebug	No ebug	No ebug
Service	State	Paused	Continue running	Paused	Continue running	Re-start then paused	Re-start then continue running	Destroyed	Destroyed
	Behaviour	Service resume whenever device is re-awake	Service continue running	Service resume whenever device is re-awake	Service continue running	Service is allowed to re-start and then resume whenever device is re-awake	Service is allowed to re-start and then continue running	Service is not allowed to re-start	Service is not allowed to re-start
	Test Case#	SHW1	SHW2	SBW1	SBW2	SSW1	SSW2	SFW1	SFW2
	Result	ebug If Service is not stopped	ebug If Service is not stopped and Wakelock is not released	ebug If Service is not stopped	ebug If Service is not stopped and Wakelock is not released	ebug If Service is not stopped	ebug If Service is not stopped and Wakelock is not released	No ebug	No ebug

A=Activity, S=Service,H=Home,B=Back,S=Swipe-out,F=Force-stop,W1=Wakelock ON,W2=Wakelock OFF

Test Case # : Activity/Service, Home/Back/Swipe/Force, Wakelock ON/OFF

Figure 5.49: Summary of results from all test cases.

5.3 Real Application Test (Aripuca GPS Tracker)

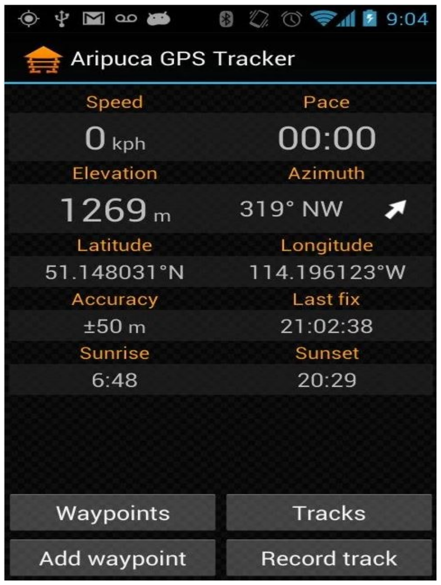
In order to verify our assumption about bounded service-related problems, we have selected an app called Aripuca GPS Tracker of version 1.3.4. It is a free and open source GPS tracking app for Android. This app provides very useful features, for example, displaying, recording, exporting/importing tracks/waypoints, tracking segments (by distance, by time, by custom intervals, etc.).

Experiment

The purpose of this experiment is to check the power consumption behaviour of the AUT especially once the app is not visible. In order to check any deviation from ideal behaviour, we have tested two versions of the same app. In the Figure 5.51, the dotted line curve represents the power consumption of the buggy app and the solid-line curve represents that of correct app. In the beginning of the test, the smartphone is in the switched off state. We start the phone by pressing the power button. After successful boot up, the

smartphone Home screen is completely visible. We do not touch the smartphones screen, and it remains in the same state for a while. At this stage, the device is in the idle state. Due to the aggressive sleeping policy implemented by the Android platform, the device goes into the sleeping mode. At this moment, we start to measure the power consumption of the device which corresponds to P1. Here, the device power consumption is 100 times less than the consumption in the On state; In order to run the test case, we press the power button of the smartphone to wake it up at time T1. The power consumption shots up from P1 to P2. Now, the display is visible, and other important components such as the CPU and radio are also in full wake up state. At this stage, no app is running on the smartphone. We measured the power consumption P2 during time interval (T1-T2), which corresponds to the app pre-execution state. At time T2, we start the AUT by clicking on the app icon. The power consumption of the device shots up from P2 to P3. The Aripuca GPS Tracker is loaded and it starts running with full functionality. Now, the app is in the execution state. We test different features of the application. At time T3 we press the button Waypoints on main screen to access Waypoints screen. This screen shows recorded tracks. We measure the power consumption P3 during the time interval (T2-T3), which corresponds to the power consumption of the app during execution. At time T3, the app is stopped by pressing the Home button from the Waypoints screen, as shown in Figure 5.51.

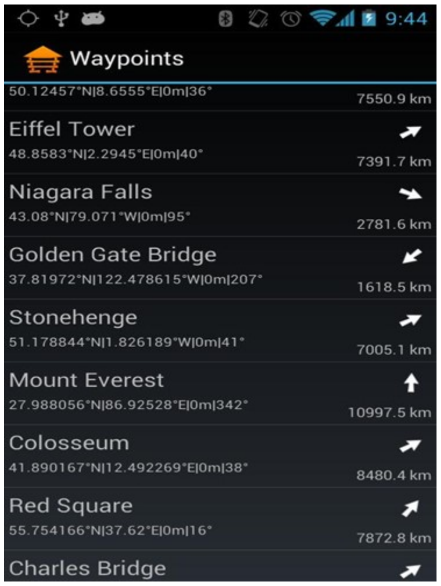
Starting from T3, the power consumption of the buggy app completely differs from that of the correct one. In the case of the correct app, the power consumption has dropped


9:04

Aripuca GPS Tracker

Speed	Pace
0 kph	00:00
Elevation	**Azimuth**
1269 m	319° NW
Latitude	**Longitude**
51.148031°N	114.196123°W
Accuracy	**Last fix**
±50 m	21:02:38
Sunrise	**Sunset**
6:48	20:29

(a) Aripuca GPS Tracker Home


9:44

Waypoints

50.12457°N|8.6555°E|0m|36°	7550.9 km
Eiffel Tower 48.8583°N|2.2945°E|0m|40°	7391.7 km
Niagara Falls 43.08°N|79.071°W|0m|95°	2781.6 km
Golden Gate Bridge 37.81972°N|122.478615°W|0m|207°	1618.5 km
Stonehenge 51.178844°N|1.826189°W|0m|41°	7005.1 km
Mount Everest 27.988056°N|86.92528°E|0m|342°	10997.5 km
Colosseum 41.890167°N|12.492269°E|0m|38°	8480.4 km
Red Square 55.754166°N|37.62°E|0m|16°	7872.8 km
Charles Bridge	

(b) Waypoints screen

Figure 5.50: Aripuca GPS tracker

from P3 to P2. Once the foreground screen is moved to the background, the AUT should be stopped. In contrast, in the case of the buggy app, the power consumption did not drop to P2 level and stayed on the P3 level. After the T3, the app is considered to be in the post-execution state. We measure the power consumption during the time interval (T3-T4) and call it the post-execution power consumption of the app. After stopping, we do not touch the device until its screen goes off automatically at time T4. After time T4, the power consumption of the correct app, it drops from P2 to P1, whereas in the case of the buggy app, it drops from P3 to P2.

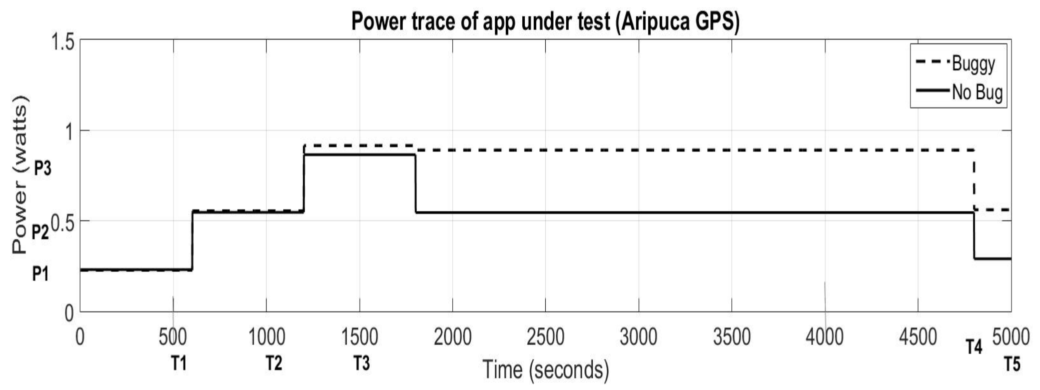


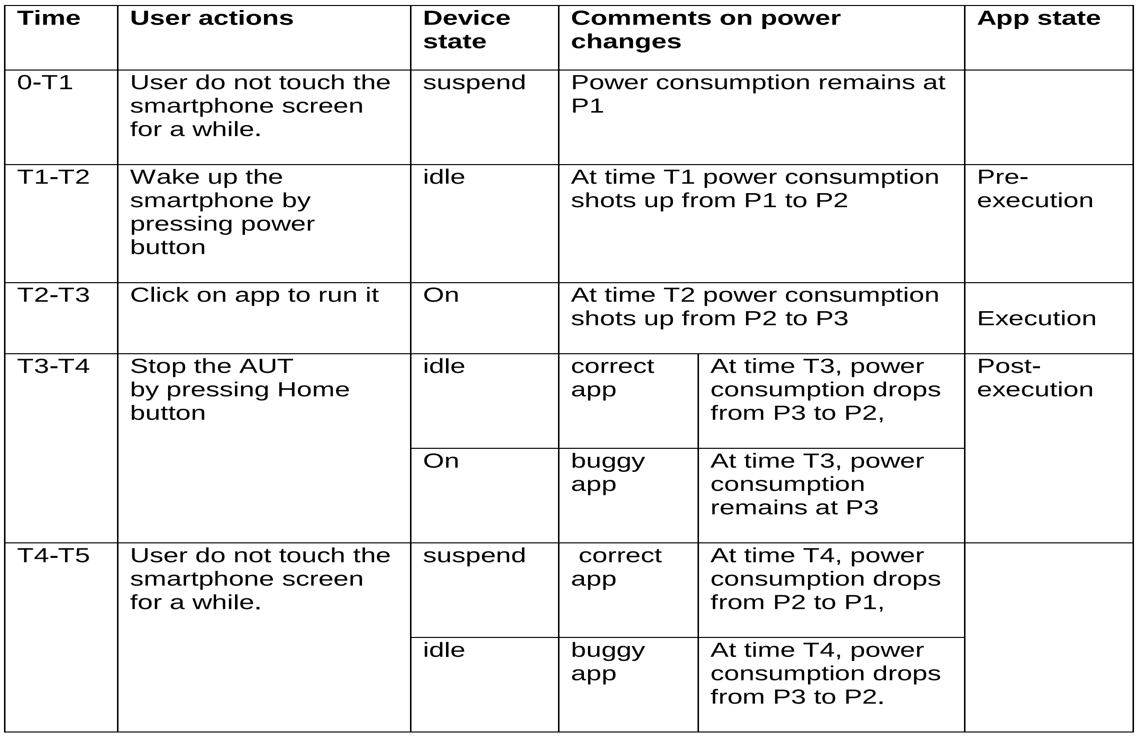
Figure 5.51: Power trace comparison of buggy and not buggy versions of Aripuca GPS Tracker

Observations

As shown in Figure 5.51, the graph shows that the power consumption of the correct app settles down to the pre-execution power state level after stopping the app, whereas in case of a buggy the app, it does not. The buggy AUT do not allow the device to go into the sleep mode, even though the display is off.

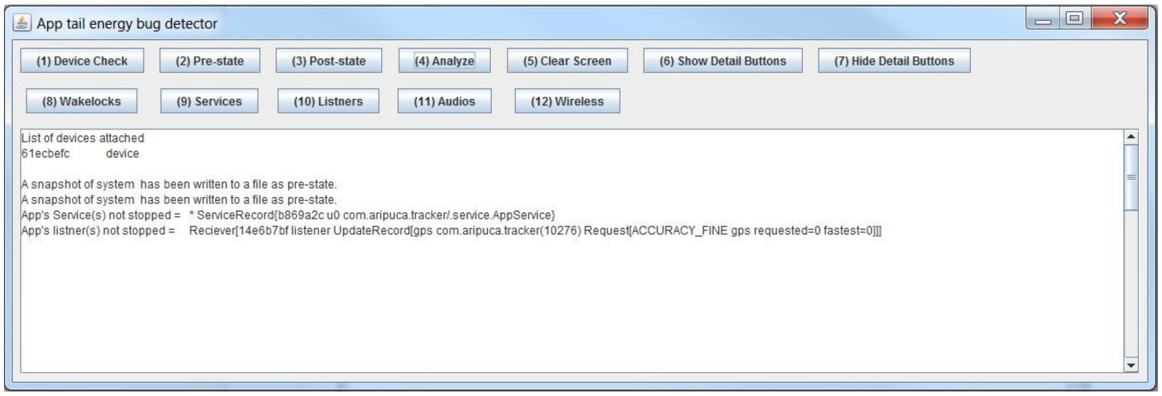
Validation In order to verify our observation and to find out the root cause of the problem, we run our application tail energy bug detector. After connecting smartphone with USB cable to developers PC, we press device check button to verify connectivity. To get pre-execution system state during the time interval (T1-T2), we press Pre-state button from our tool as shown in Figure 5.53. Later to get post-execution system state during time interval (T4-T5), we press Post-state button. To find out the difference between pre and post state files, we press Analyze button. The output of buggy and correct apps are shown in Figure 5.53 and Figure 5.54 respectively. For simplicity, detail button palette is not visible. To show it press show detail buttons and to hide again press hide details button. To check what is still active even after closing the app, we press any details button such as wakelocks, services etc.

The output shows a service named Service.AppService is still running. Second a listener is also active. After pressing the home button listener should must be released otherwise it can continuously keeping location services active. In correct app, no listener is active. Though Service.AppService is active but not consuming battery power.



Time	User actions	Device state	Comments on power changes		App state
0-T1	User do not touch the smartphone screen for a while.	suspend	Power consumption remains at P1		
T1-T2	Wake up the smartphone by pressing power button	idle	At time T1 power consumption shots up from P1 to P2		Pre-execution
T2-T3	Click on app to run it	On	At time T2 power consumption shots up from P2 to P3		Execution
T3-T4	Stop the AUT by pressing Home button	idle	correct app	At time T3, power consumption drops from P3 to P2,	Post-execution
		On	buggy app	At time T3, power consumption remains at P3	
T4-T5	User do not touch the smartphone screen for a while.	suspend	correct app	At time T4, power consumption drops from P2 to P1,	
		idle	buggy app	At time T4, power consumption drops from P3 to P2.	

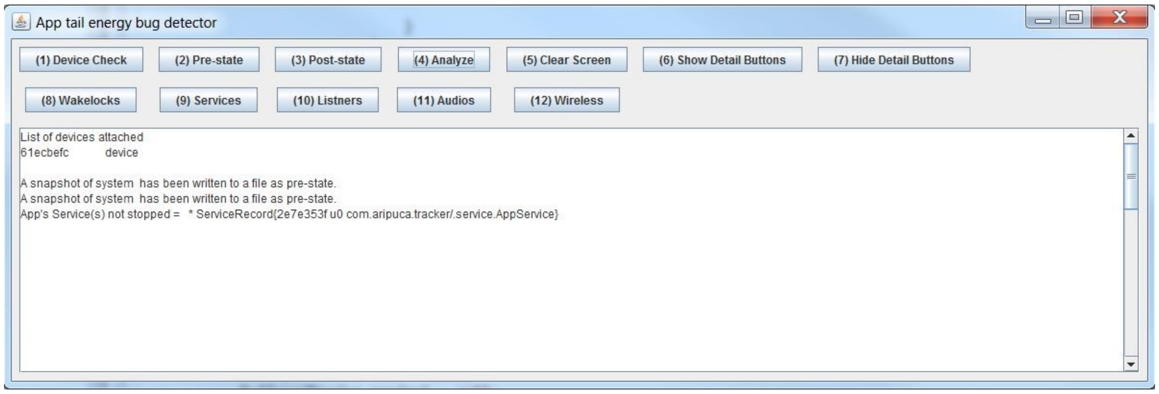
Figure 5.52: Summary of the main states of the AUT during test case execution



```
List of devices attached
61ecbefc        device

A snapshot of system  has been written to a file as pre-state.
A snapshot of system  has been written to a file as pre-state.
App's Service(s) not stopped =   * ServiceRecord{b869a2c u0 com.aripuca.tracker/.service.AppService}
App's listner(s) not stopped =    Reciever[14e6b7bf listener UpdateRecord[gps com.aripuca.tracker(10276) Request[ACCURACY_FINE gps requested=0 fastest=0]]]
```

Figure 5.53: Result from application tail energy bug tool (Aripuca GPS tracker "buggy app").



```
List of devices attached
61ecbefc        device

A snapshot of system  has been written to a file as pre-state.
A snapshot of system  has been written to a file as pre-state.
App's Service(s) not stopped =   * ServiceRecord{2e7e353f u0 com.aripuca.tracker/.service.AppService}
```

Figure 5.54: Result from application tail energy bug tool (Aripuca GPS tracker "correct app").

Experiment outcome and discussion

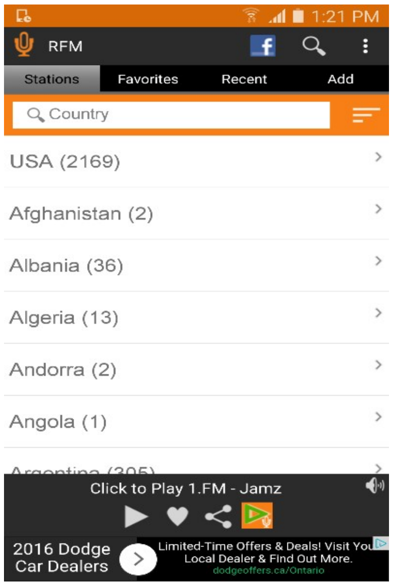
We conducted an experiment to show the presence of the app tail energy bug when services are not handled properly. First, using monsoon power meter we discovered a reasonable difference in power consumption before and after closing the app which shows an indication of tail energy bug. To validate it, we used our tool application tail energy bug detector. The tool compares pre and post system states and showed any possible difference. For Aripuca GPS tracker app, our tool displayed a list of services and listeners causing continuous battery drain.

5.4 Real Application Test (FM Radio)

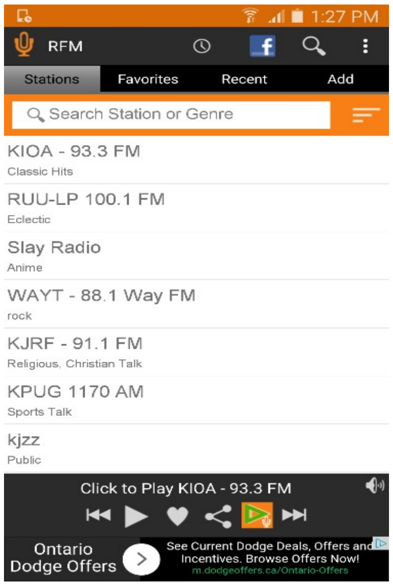
In order to verify our assumption that mismatch between user requirements and developer understanding leads to app-tail-energy bug in smartphone. User are cautious about phone battery that is why they want to open only one app at a time. Users try different ways to stop the app like pressing Home button and swiping the app out from stack etc. In this experiment, we are exploring the most suitable and efficient procedure to close an app. We have selected Radio FM. It is an app to play radio on mobile phones online. It is a live, Internet based radio service. Radio FM allows you to listen and enjoy variety of programs like songs, music, talks, news, comedy, shows, and concerts.

Experiment

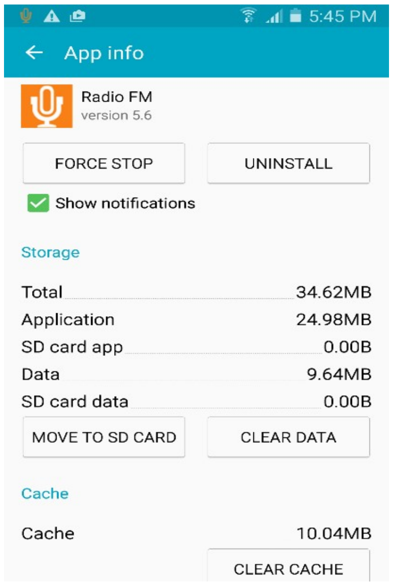
The purpose of this experiment is to check the power consumption behaviour of the AUT especially once the app is not visible or closed. In order to check any deviation from ideal behaviour, we have run two test cases of the same app. In the Figure 5.56, the dotted line curve represents the power consumption of the first test case which shows buggy behaviour and the solid-line curve represents second test case that of correct behaviour. In the beginning of the test, the smartphone is in the switched off state. We start the phone by pressing the power button. After successful boot up, the smartphone Home screen is completely visible. In this stage, the device is in the idle state. Due to the aggressive sleeping policy implemented by the Android platform, the device goes into the sleeping mode. At this moment, we start to measure the power consumption of the device which corresponds to P1. Here, the device power consumption is 100 times less than the consumption in the On state. In order to run the test case, we press the power button of the smartphone to wake it up at time T1. The power consumption shots up from P1 to P3. Now, the display is visible, and other important components such as the CPU and radio are also in full wake up state. At this stage, no app is running on the smartphone. We measured the power consumption P3 during time interval (T1-T2), which corresponds to the app pre-execution state. At time T2, we start the AUT by clicking on the app icon. The power consumption of the device shots up from P3 to P4. The FM Radio app is loaded and it starts running with full functionality. Now, the app is in the execution state. We test different features of the application. At Time T3, choose and click on any radio station to play it. At T4 continue running radio. We measure power consumption P5 because we continue playing radio and do not stop it even after time T4. At time T5, in the first test case, the app is stopped by pressing the Home button from the RFM stations (search stations or genre) screen, as shown in Figure 5.56, the current screen goes into background. User can optionally swipe out the radio activity from recent running applications list. In the second case, app is stopped by pressing force stop from settings.


1:21 PM
RFM
Stations | Favorites | Recent | Add
Country
USA (2169)
Afghanistan (2)
Albania (36)
Algeria (13)
Andorra (2)
Angola (1)
Argentina (305)
Click to Play 1.FM - Jamz
2016 Dodge Car Dealers
Limited-Time Offers & Deals! Visit Your Local Dealer & Find Out More.
dodgeoffers.ca/Ontario

(a) FM Radio Main


1:27 PM
RFM
Stations | Favorites | Recent | Add
Search Station or Genre
KIOA - 93.3 FM
Classic Hits
RUU-LP 100.1 FM
Eclectic
Slay Radio
Anime
WAYT - 88.1 Way FM
rock
KJRF - 91.1 FM
Religious, Christian Talk
KPUG 1170 AM
Sports Talk
kjzz
Public
Click to Play KIOA - 93.3 FM
Ontario Dodge Offers
See Current Dodge Deals, Offers and Incentives. Browse Offers Now!
m.dodgeoffers.ca/Ontario-Offers

(b) FM Radio Genre


5:45 PM
App info
Radio FM
version 5.6
FORCE STOP | UNINSTALL
Show notifications
Storage

Total	34.62MB
Application	24.98MB
SD card app	0.00B
Data	9.64MB
SD card data	0.00B

MOVE TO SD CARD | CLEAR DATA
Cache
Cache 10.04MB
CLEAR CACHE

(c) Force Stop

Figure 5.55: FM Radio App

Once the foreground screen is moved to the background, the AUT should be stopped including started service as well. Starting from T5, the power consumption of the app in the absence of background started service is completely differs from that of the presence of started service. In the absence of started service, the app power consumption has dropped from P5 to P3. In contrast, in the presence of the started service, the app power consumption did not drop to P3 level but stay above P3 level. After the T5, the app is considered to be in the post-execution state. We measure the power consumption during the time interval (T5-T6) and call it the post-execution power consumption of the app. After stopping, we do not touch the device until its screen goes off automatically. At time T6, in first test case, the power consumption of the app with a started service, drops from P3 to P2 and stays at P2 continuously, but during second case, the power consumption of the app without a started service, drops from P3 to P1 and device goes into suspend (sleep) mode.

Steps S1-S5, S6a-S7 correspond to the test represented by solid line (correct behaviour) Steps S1-S5, S6b-S7 correspond to the test represented by dotted line (buggy behaviour)

Observations

As shown in Figure 5.56, the graph shows that power consumption does not settle

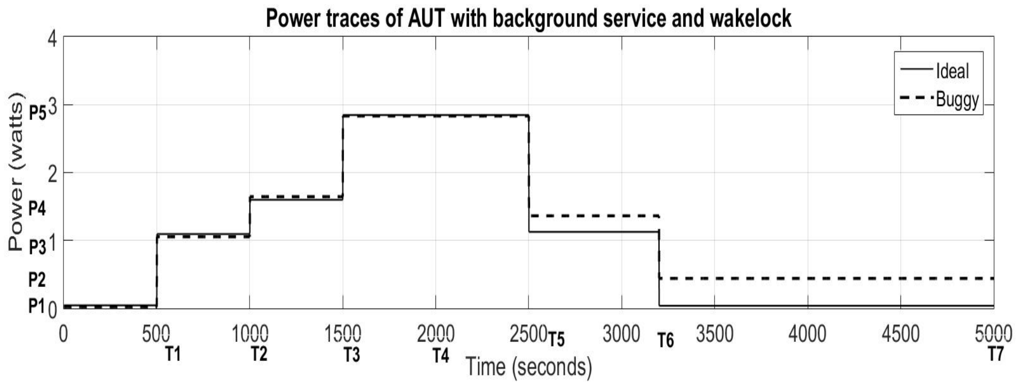
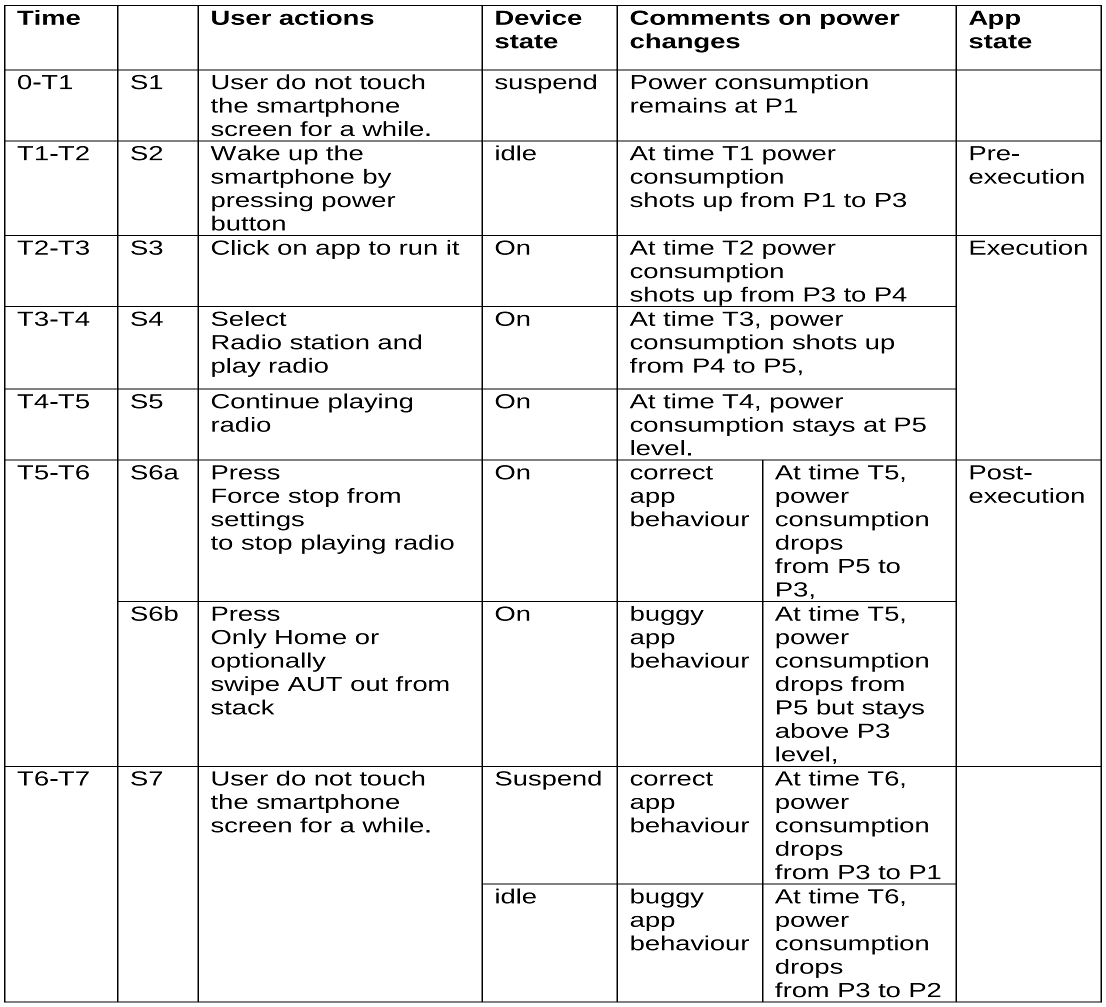


Figure 5.56: Power trace comparison of buggy and not buggy versions of FMRadio App

down to the pre-execution power state level after stopping the AUT when Home buttons is pressed or swiping the AUT out from stack. We observed the radio is still playing which means app is not yet closed completely and that is why device does not go into the suspend mode. The device goes into suspend mode after a while when app is completely close by using OS provided force stop button from settings. We assume that this button is properly implemented by Android OS

Experiment outcome and discussion

We conducted an experiment to show the presence of the app-tail-ebug when user applied only home button to close the app which is not sufficient due to running background media service.



Time		User actions	Device state	Comments on power changes		App state
0-T1	S1	User do not touch the smartphone screen for a while.	suspend	Power consumption remains at P1		
T1-T2	S2	Wake up the smartphone by pressing power button	idle	At time T1 power consumption shots up from P1 to P3		Pre-execution
T2-T3	S3	Click on app to run it	On	At time T2 power consumption shots up from P3 to P4		Execution
T3-T4	S4	Select Radio station and play radio	On	At time T3, power consumption shots up from P4 to P5,		
T4-T5	S5	Continue playing radio	On	At time T4, power consumption stays at P5 level.		
T5-T6	S6a	Press Force stop from settings to stop playing radio	On	correct app behaviour	At time T5, power consumption drops from P5 to P3,	Post-execution
	S6b	Press Only Home or optionally swipe AUT out from stack	On	buggy app behaviour	At time T5, power consumption drops from P5 but stays above P3 level,	
T6-T7	S7	User do not touch the smartphone screen for a while.	Suspend	correct app behaviour	At time T6, power consumption drops from P3 to P1	
			idle	buggy app behaviour	At time T6, power consumption drops from P3 to P2	

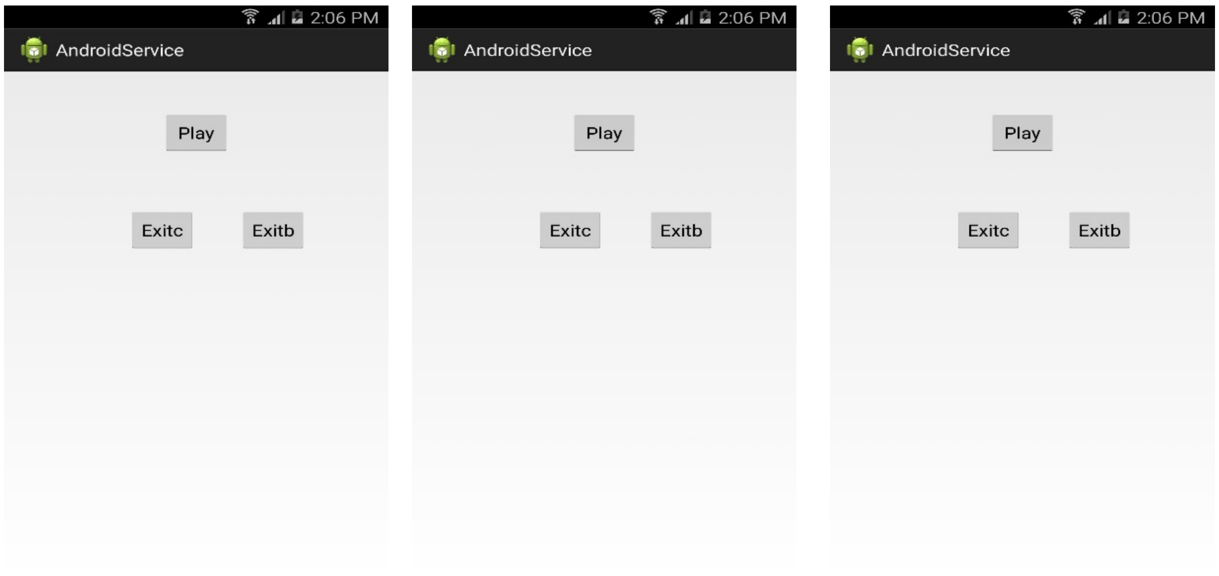
Figure 5.57: Summary of the main states of the AUT during test case execution

5.5 Real Application Test (AndriodExit Demo App)

In order to verify our assumption that coding error behind exit button leads to app-tail-energy bug in smartphone. User are cautious about phone battery thats why they want to open only one app at time. In order to close an app completely, often user wants a single click operation. In this experiment, we are comparing the effect of coding errors behind exit button. We have designed a demo app with a play music button and two exit buttons. First exit button provides clean exit while the other one buggy exit behaviour.

Experiment

The purpose of this experiment is to check the power consumption behaviour of the AUT especially once the app is not visible or closed. In order to check any deviation from ideal behaviour, we have run two test cases of the same app. In the Figure 9.2, the dotted line curve represents the power consumption of the first test case which shows buggy behaviour and the solid-line curve represents second test case that of correct behaviour. In the beginning of the test, the smartphone is in the switched off state. We start the phone by pressing the power button. After successful boot up, the smartphone Home screen is completely visible. In this stage, the device is in the idle state. Due to the aggressive sleeping policy implemented by the Android platform, the device goes into the sleeping mode. At this moment, we start to measure the power consumption of the device which corresponds to P1. Here, the device power consumption is 100 times less than the consumption in the On state. In order to run the test case, we press the power button of the smartphone to wake it up at time T1. The power consumption shots up from P1 to P3. Now, the display is visible, and other important components such as the CPU and radio are also in full wake up state. At this stage, no app is running on the smartphone. We measured the power consumption P3 during time interval (T1-T2), which corresponds to the app pre-execution state. At time T2, we start the AUT by clicking on the app icon. The power consumption of the device shots up from P3 to P4. The AndriodExit demo app is loaded and it starts running with full functionality. Now, the app is in the execution state. We test different features of the application. At Time T3, choose and click on play button to play the music file. At T4 continue running music. We measure power consumption P5 because we continue playing music and do not stop it even after time T4. At time T5, in the first test case, the app is stopped by pressing the Exitb button from the AndriodService screen, as shown in Figure 9.1, the current screen goes into background. In the second case, app is stopped by pressing Exitc button. Once the foreground screen is moved to the background, the AUT should be stopped including started service as well. Starting from T5, the power consumption of the app in the absence of background started service is completely differs from that of the presence of started service. In the absence of



(a) Exit1 (b) Exit2 (c) Exit3

Figure 5.58: AndroidExit demo app

started service, the app power consumption has dropped from P5 to P3. In contrast, in the presence of the started service, the app power consumption did not drop to P3 level but stay above P3 level. After the T5, the app is considered to be in the post-execution state. We measure the power consumption during the time interval (T5-T6) and call it the post-execution power consumption of the app. After stopping, we do not touch the device until its screen goes off automatically. At time T6, in first test case, the power consumption of the app with a started service, drops from P3 to P2 and stays at P2 continuously, but during second case, the power consumption of the app without a started service, drops from P3 to P1 and device goes into suspend (sleep) mode.

Steps S1-S5, S6a-S7 correspond to the test represented by solid line (correct behaviour) Steps S1-S5, S6b-S7 correspond to the test represented by dotted line (buggy behaviour)

Observations

As shown in Figure 5.59, the graph shows that power consumption does not settle down to the pre-execution power state level after stopping the AUT when buggy exit (Exitb) button is pressed. We observed the music is still playing which means app is not yet closed completely and thats why device does not go into the suspend mode. The device goes into suspend mode after a while when app is completely close by using clean exit (Exitc)

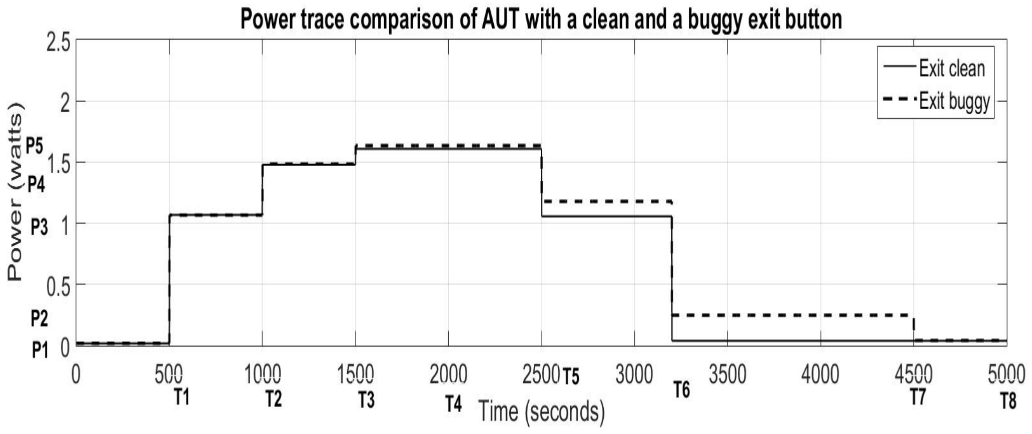


Figure 5.59: Power trace comparison of AndroidExit demo app

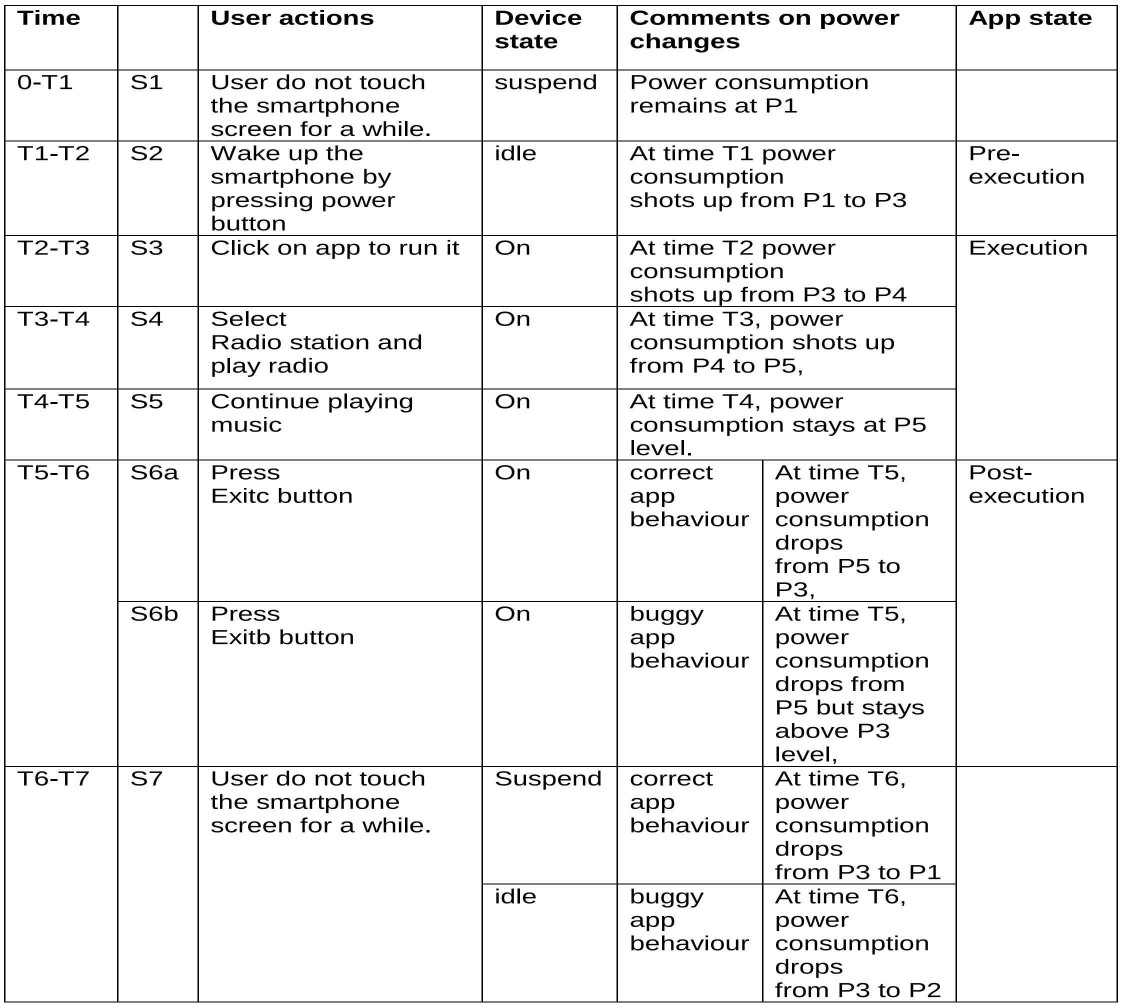
button from settings.

Experiment outcome and discussion

We conducted an experiment to show the presence of the app-tail-ebug when due to coding errors behind exit button, app is not completely closed.

5.6 Summary

In summary, it does not really matter whether you use back or home: it only changes what the app shows you next time you run it. It does not have an effect on battery use. Neither of them corresponds to ”exiting” a program on your PC.



Time		User actions	Device state	Comments on power changes		App state
0-T1	S1	User do not touch the smartphone screen for a while.	suspend	Power consumption remains at P1		
T1-T2	S2	Wake up the smartphone by pressing power button	idle	At time T1 power consumption shots up from P1 to P3		Pre-execution
T2-T3	S3	Click on app to run it	On	At time T2 power consumption shots up from P3 to P4		Execution
T3-T4	S4	Select Radio station and play radio	On	At time T3, power consumption shots up from P4 to P5,		
T4-T5	S5	Continue playing music	On	At time T4, power consumption stays at P5 level.		
T5-T6	S6a	Press Exitc button	On	correct app behaviour	At time T5, power consumption drops from P5 to P3,	Post-execution
	S6b	Press Exitb button	On	buggy app behaviour	At time T5, power consumption drops from P5 but stays above P3 level,	
T6-T7	S7	User do not touch the smartphone screen for a while.	Suspend	correct app behaviour	At time T6, power consumption drops from P3 to P1	
			idle	buggy app behaviour	At time T6, power consumption drops from P3 to P2	

Figure 5.60: Summary of the main states of the AUT during test case execution

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We focused on energy bugs in smartphones. To provide a clear understanding of energy bugs, we developed an operational definition that could be easily translated into a procedure to detect the presence of energy bugs. Furthermore, we integrated the proposed definition in a diagnostic procedure to facilitate the application of the proposed definition. We investigated the existence of energy bugs when apps or platforms were updated. We validated the proposed definition with measurements and real-world energy bug examples. Results showed that there were energy bugs across different versions of the same app as well as across different versions of the same platform.

A class of abnormal battery drain in smartphones comes from the misuse of OS power management facility, services, and specific user actions. After application deployment, the only way to handle these energy bugs is to perform a runtime check and correction against faulty applications. In this work, we took Android based system as our primary target, and focused on the WakeLock facility, which is the cornerstone of Android Power Management framework. The main contribution of this work is a runtime WakeLock bug detection framework. This framework is implemented in a tool, "app-tail-ebug detector" designed by us. This tool helps user detect energy bugs due to remaining background app's components when app is not running. Moreover, this user-level tool does not need hacking of the OS kernel and recompilation, as some related works did.

For in-depth system monitoring, we utilized ADB (Android debug bridge) commands provided by Google and Android framework, which is a suite of various manager services, such as activity manager, power manager etc. ADB provides valuable info about "Activity" and "Service", while Android framework keeps track of wakelocks. The PowerManagerService class of Android framework offers internal function to dump current wakelock data which cannot be called by the user. However, we can obtain the same wakelock data via ADB shell command: "adb shell dumpsys power". To check the behaviour of Android app components such as "Activities" and "Services" with or without wakelock, we developed a demo app with two screens. The first screen has buttons to acquire or release four different types of wakelocks. The second screen provides buttons to start and stop a service with built-in four types of wakelocks. Our results showed that activities cannot run in the background in any case either with or without a wakelock. On the other hand, Services can run in the background. It stops when the device goes to sleep and resume whenever the device re-awakes. Sometime, based on user requirements, It is not useful to resume the task which starts before the device went to sleep. It is a waste and for this reason we call it application tail energy bug. In the presence of wakelock when the device is forced to stay awake, "Service" is allowed to run continuously in the background till it finishes intended task. The user is not aware of what is going on inside smartphone, but high battery temperature increases his or her suspicion. For example, persistent partial wakelock taken by radio app can deplete the smartphone battery within few hours. Using Monsoon power meter, we measured the power consumption before and after closing the application under test (AUT). A significant difference in power consumption might be an indication of an ebug. To verify the presence of ebug and find out the root cause of the problem, we run our tool, which is based on ADB commands. It provides real-time status of AUT components and wakelocks.

6.2 Future Work

The proposed framework utilized the black-box approach for testing energy bugs. In future, we are interested in integrating the proposed definition in a white-box testing framework. In addition, we intend to implement details of detecting resource leaks, wakelock bug, vacuous background services bugs, immortality bugs, and suboptimal resource binding bugs, tail energy bugs, expensive background services bugs, and loop energy bugs modules in the proposed framework. We are interested to fully implement the framework and test it using different types of apps. We deployed our tool on a smartphone as an app. However, we intend to deploy our tool as an app on all versions of Android OS including Android-4.4 KitKat and above.

Bibliography

[1] "Apple confirms battery life problems are ios 5 related," http://www.wired.com/2011/11/iphone-4s-battery-issues/, Nov 2011.

[2] "Possible issue leading to iphone 4s battery drain," https://discussions.apple.com/thread/3491965, November 2011.

[3] "Wakelocks: Detect no-sleep issues in android* applications," https://software.intel.com/en-us/android/articles/wakelocks-detect-nosleep- issues-in-android-applications, June 2013.

[4] "Android powermanagerservice," https://github.com/android/platform-frameworks-base/blob/master/services/core/java/com/android/server/power/ PowerManagerService.java, January 2014.

[5] "Faulty proximity sensor in galaxy note 3," https://community.verizonwireless.com/thread/814958, January 2014.

[6] "Global mobile statistics 2014: Mobile subscribers; handset market share; mobile operators," http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats, May 2014.

[7] "Activity," https://developer.android.com/guide/components/activities.html, Nov 2015.

[8] "Services," https://developer.android.com/guide/components/services.html, Nov 2015.

[9] A. Abogharaf, R. Palit, K. Naik, and A. Singh, "A methodology for energy performance testing of smartphone applications," in *Automation of Software Test (AST), 2012 7th Int. Workshop on.* IEEE, 2012, pp. 110–116.

[10] A. A. Albasir and K. Naik, "Smow: An energy-bandwidth aware web browsing technique for smartphones," *Access, IEEE*, vol. 2, pp. 1427–1441, 2014.

[11] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proc. of the 22nd ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, 2014, pp. 588–598.

[12] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Software Engineering (ICSE), 2013 35th Int. Conf. on.* IEEE, 2013, pp. 92–101.

[13] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, "Devscope: a nonintrusive and online power analysis tool for smartphone hardware components," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* ACM, 2012, pp. 353–362.

[14] K. Kim and H. Cha, "Wakescope: runtime wakelock anomaly management scheme for android platform," in *Proceedings of the Eleventh ACM International Conference on Embedded Software.* IEEE Press, 2013, p. 27.

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

[16] Y. Liu, C. Xu, S. C. Cheung, and J. Lu, "Greendroid: automated diagnosis of energy inefficiency for smartphone applications," *Software Engineering, IEEE Transactions on*, vol. 40, no. 9, pp. 911–940, 2014.

[17] 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 *NSDI*, vol. 13, 2013, pp. 57–70.

[18] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica, "Collaborative energy debugging for mobile devices," in *Proc. of the Eighth USENIX conf. on Hot Topics in System Dependability*, 2012, pp. 6–6.

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

[20] ——, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proc. of the 7th ACM european conf. on Computer Systems*, 2012, pp. 29–42.

[21] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, “Fine-grained power modeling for smartphones using system call tracing,” in *Proceedings of the sixth conference on Computer systems.* ACM, 2011, pp. 153–168.

[22] 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. of the 10th int. conf. on Mobile systems, applications, and services.* ACM, 2012, pp. 267–280.

[23] R. Powers *et al.*, “Batteries for low power electronics,” *Proc. of the IEEE*, vol. 83, no. 4, pp. 687–693, 1995.

[24] J. Singh, V. Mahinthan, and K. Naik, “Automation of energy performance evaluation of software applications on servers,” in *Proc. of SERP*, vol. 14, p. 7.

[25] S. Tarkoma, M. Siekkinen, E. Lagerspetz, and Y. Xiao, *Smartphone energy consumption: modeling and optimization*, First Cambridge University Press ed., 2014.

[26] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, “Towards verifying android apps for the absence of no-sleep energy bugs,” in *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*, 2012.

[27] K. Yaghmour, *Embedded Android*, First Edition ed., 2013.

[28] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, “Appscope: Application energy metering framework for android smartphone using kernel activity monitoring,” in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 387–400.

[29] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang, “Adel: An automatic detector of energy leaks for smartphone applications,” in *Proceedings of the eighth IEEE/ACM/IFIP int. conf. on Hardware/software codesign and system synthesis*, 2012, pp. 363–372.

[30] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, “Accurate online power estimation and automatic battery behavior based power model generation for smartphones,” in *Proc. of the eighth IEEE/ACM/IFIP int. conf. on Hardware/software codesign and system synthesis*, 2010, pp. 105–114.

Appendix

In the following section, I present memory dump of power manager, which is keeping track of wakelocks data. A list of total number of wakelocks and their details can be found between

Wake Locks: size=0 and Suspended Blockers: size=4 (page=107)

```
Android Debug Bridge (ADB) commands
How to check active wakelocks
 adb shell dumpsys power
Power Manager State:
  mDirty=0x0
  mWakefulness=Asleep
  mInteractive=false
  mIsPowered=false
  mPlugType=0
  mBatteryLevel=13
  mBatteryLevelWhenDreamStarted=0
  mDockState=0
  mStayOn=false
  mProximityPositive=false
  mBootCompleted=true
  mSystemReady=true
  mHalAutoSuspendModeEnabled=true
  mHalInteractiveModeEnabled=false
  mWakeLockSummary=0x0
  mUserActivitySummary=0x0
  mRequestWaitForNegativeProximity=false
  mSandmanScheduled=false
  mSandmanSummoned=false
  mLowPowerModeEnabled=false
  mBatteryLevelLow=true
  mLastWakeTime=30779 (757509 ms ago)
  mLastSleepTime=772273 (16015 ms ago)
  mLastUserActivityTime=757266 (31022 ms ago)
  mLastUserActivityTimeNoChangeLights=37581 (750707 ms ago)
  mLastInteractivePowerHintTime=775374 (12914 ms ago)
  mDisplayReady=true
  mHoldingWakeLockSuspendBlocker=false
  mHoldingDisplaySuspendBlocker=false
Settings and Configuration:
  mDecoupleHalAutoSuspendModeFromDisplayConfig=false
  mDecoupleHalInteractiveModeFromDisplayConfig=false
  mWakeUpWhenPluggedOrUnpluggedConfig=true
  mSuspendWhenScreenOffDueToProximityConfig=false
  mDreamsSupportedConfig=true
  mDreamsEnabledByDefaultConfig=true
  mDreamsActivatedOnSleepByDefaultConfig=false
  mDreamsActivatedOnDockByDefaultConfig=true
  mDreamsEnabledOnBatteryConfig=false
  mDreamsBatteryLevelMinimumWhenPoweredConfig=-1
  mDreamsBatteryLevelMinimumWhenNotPoweredConfig=15
  mDreamsBatteryLevelDrainCutoffConfig=5
  mDreamsEnabledSetting=false
  mDreamsActivateOnSleepSetting=false
  mDreamsActivateOnDockSetting=true
  mDozeAfterScreenOffConfig=false
  mLowPowerModeSetting=false
  mAutoLowPowerModeConfigured=false
  mAutoLowPowerModeSnoozing=false
  mMinimumScreenOffTimeoutConfig=3000
  mMaximumScreenDimDurationConfig=7000
  mMaximumScreenDimRatioConfig=0.20000005
  mScreenOffTimeoutSetting=15000
  mSleepTimeoutSetting=-1
  mMaximumScreenOffTimeoutFromDeviceAdmin=2147483647 (enforced=false)
  mStayOnWhilePluggedInSetting=0
  mScreenBrightnessSetting=255
  mScreenAutoBrightnessAdjustmentSetting=0.0
  mScreenBrightnessModeSetting=0
  mScreenBrightnessOverrideFromWindowManager=-1
  mUserActivityTimeoutOverrideFromWindowManager=5000
  mTemporaryScreenBrightnessSettingOverride=-1
  mTemporaryScreenAutoBrightnessAdjustmentSettingOverride=NaN
  mDozeScreenStateOverrideFromDreamManager=0
  mDozeScreenBrightnessOverrideFromDreamManager=-1
  mScreenBrightnessSettingMinimum=5
  mScreenBrightnessSettingMaximum=255
  mScreenBrightnessSettingDefault=162
  mLastUserActivitySummary: 0
  SecProductFeature_FRAMEWORK.
  SEC_PRODUCT_FEATURE_FRAMEWORK_ENABLE_SMART_STAY: true
  SecProductFeature_CAMERA.
  SEC_PRODUCT_FEATURE_CAMERA_DELAY_TIME_FOR_SMART_STAY: 275
Sleep timeout: -1 ms
Screen off timeout: 5000 ms
Screen dim duration: 0 ms
Screen dim duration override: 0 ms
Smart Stay:
  USE_SMART_STAY: true
  USE_PRE_SMART_STAY: false
  mSmartStayEnabledSetting: true
  SmartStayDelay: 2750
  mNextTimeoutForSmartStay: 766516
  mPendingMessageSmartStay: false
  mPendingMessagePreSmartStay: false
  mFaceDetected: false
  mIsBadCurrentConsumptionDevice: true
InputDeviceLightState:
  mTouchKeyOffTimeoutSetting: 1500
  mIsSipVisible: false
  mTouchKeyForceDisable: false
  mPowerSaveModeSettingBroadcasted: false
  mLimitedPerformanceBroadcasted: false
  mTouchKeyForceDisableOverrideFromSystemPowerSaveMode: false
  mEmergencyMode: false
Clear Cover:
  mIsCoverClosed: false
  mIsReadyCoverFromPWM: false
  mFeatureCoverSysfs: true
  mIsCocktailBarCover: false
ALPM Mode:
  mIsForceUnblankDisplay: false
  mIsAlpmMode: false
```

```
Wake Locks: size=0

Suspend Blockers: size=4
  PowerManagerService.WakeLocks: ref count=0
  PowerManagerService.Display: ref count=0
  PowerManagerService.Broadcasts: ref count=0
  PowerManagerService.WirelessChargerDetector: ref count=0
Display Power: state=OFF
Wireless Charger Detector State:
  mGravitySensor={Sensor name="Gravity Sensor", vendor="Samsung Electronics",

   version=3, type=9, maxRange=19.6133, resolution=5.9604645E-8, power=6.0,

  minDelay=10000}

  mPoweredWirelessly=false
  mAtRest=false
  mRestX=0.0, mRestY=0.0, mRestZ=0.0
  mDetectionInProgress=false
  mDetectionStartTime=0 (never)
  mMustUpdateRestPosition=false
  mTotalSamples=0
  mMovingSamples=0
  mFirstSampleX=0.0, mFirstSampleY=0.0, mFirstSampleZ=0.0
  mLastSampleX=0.0, mLastSampleY=0.0, mLastSampleZ=0.0
```