How Do Developers Use Wake Locks in Android Applications? A Large-Scale Empirical Study

Technical Report HKUST-CS15-04
Online date: November 13, 2015

Abstract: Wake locks are commonly used in Android apps to protect critical computations from being disrupted by device sleeping. However, inappropriate use of wake locks often causes various issues that can seriously impact app user experience. Unfortunately, people have limited understanding of how Android developers use wake locks in practice and the potential issues that can arise from wake lock misuses. To bridge this gap, we conducted a large-scale empirical study on 1.1 million commercial and 31 open-source Android apps. By automated program analysis of commercial apps and manual investigation of the bug reports and source code revisions of open-source apps, we made several important findings. For example, we found that developers often use wake locks to protect 15 types of computational tasks that can bring users observable or perceptible benefits. We also identified eight types of wake lock misuses that commonly cause functional or non-functional issues, only two of which had been studied by existing work. Such empirical findings can provide guidance to developers on how to appropriately use wake locks and shed light on future research on designing effective techniques to avoid, detect, and debug wake lock issues.

Authors:

Yepang Liu (Hong Kong University of Science and Technology)

Chang Xu (Nanjing University)

Shing-Chi Cheung (Hong Kong University of Science and Technology)

Valerio Terragni (Hong Kong University of Science and Technology)

How Do Developers Use Wake Locks in Android Apps? A Large-scale Empirical Study

Yepang Liu[§], Chang Xu[‡], Shing-Chi Cheung[§], and Valerio Terragni[§]
[§] Dept. of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China
[‡] State Key Lab for Novel Software Technology and Dept. of Computer Sci. and Tech., Nanjing University, Nanjing, China
[§] {andrewust, scc, vterragni}@cse.ust.hk, [‡] changxu@nju.edu.cn

ABSTRACT

Wake locks are commonly used in Android apps to protect critical computations from being disrupted by device sleeping. However, inappropriate use of wake locks often causes various issues that can seriously impact app user experience. Unfortunately, people have limited understanding of how Android developers use wake locks in practice and the potential issues that can arise from wake lock misuses. To bridge this gap, we conducted a large-scale empirical study on 1.1 million commercial and 31 open-source Android apps. By automated program analysis of commercial apps and manual investigation of the bug reports and source code revisions of open-source apps, we made several important findings. For example, we found that developers often use wake locks to protect 15 types of computational tasks that can bring users observable or perceptible benefits. We also identified eight types of wake lock misuses that commonly cause functional or non-functional issues, only two of which had been studied by existing work. Such empirical findings can provide guidance to developers on how to appropriately use wake locks and shed light on future research on designing effective techniques to avoid, detect, and debug wake lock issues.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: [Testing and Debugging]

General Terms

Experimentation, Performance

Keywords

Wake lock, critical computation, wake lock misuse

1. INTRODUCTION

Nowadays, smartphones are equipped with powerful hardware such as HD screen and various sensors to provide rich user experience. However, such hardware is a big consumer of battery power. To prolong battery life, many smartphone platforms such as Android choose to put energy-consumptive hardware into an idle or sleep mode (e.g., turning screen off) after a short period of user inactivity [35].

Albeit preserving energy, this sleeping policy may break the functionality of some apps that need to keep smartphones awake for certain critical computation. Consider a banking app. When its user transfers money online over slow network connections, it may take a while for the transaction to complete. If the user's smartphone falls asleep while waiting for server messages and does not respond in time, the transaction will fail, causing bad user experience. To address this problem, modern smartphone platforms allow apps to explicitly control whether to keep certain hardware awake for continuous computation. On Android platforms, *wake locks* are designed for this purpose. Specifically, to keep certain hardware awake for computation, an app needs to acquire an appropriate type of wake lock from the Android OS (see Section 2.2). When the computation completes, the app should release the acquired wake locks properly.

The wake lock mechanism is widely used in practice. We found that around 27.2% of apps on Google Play store [14] use wake locks for reliably providing certain functionalities (see Section 3). Despite the popularity, programming wake locks correctly is a non-trivial task. In order to avoid undesirable consequences, a cautious developer should carefully think through at least the following three questions before they use wake locks:

1. *Do the benefits brought by using wake locks justify the energy cost (for reasoning about the necessity of using wake locks)?*
2. *Which hardware needs to stay awake (for deciding which type of wake lock to use)?*
3. *When should the hardware be kept awake and when are they allowed to fall asleep (for deciding the program points to acquire and release wake locks)?*

Unfortunately, we observe that in practice, many developers seem to be unaware of such questions and use wake locks in a undisciplined way. For example, our investigation of 31 popular open-source Android apps revealed that 19 (61.3%) of them have suffered from various functional and non-functional issues/bugs due to wake lock misuses. These issues caused a lot of user frustrations. Yet, existing work, including our earlier one, only studied a small fraction of them [24, 30, 35]. Developers still lack guidance on how to appropriately use wake locks and have limited access to useful tools that can help catch their mistakes. This motivates us to conduct an empirical study to understand the common practices of wake lock usage in reality (**our first goal**) as well as common misuses of wake locks (**our second goal**).

In this paper, we study a large number of real-world Android apps, including 1,117,195 commercial ones and 31 open-source ones, by leveraging static program analysis and statistical analysis techniques. Our study aims to answer the following five research questions:

- **RQ1 (Locking app components):** *In which types of app components do Android apps often acquire wake locks?*
- **RQ2 (Lock types):** *What types of wake locks are commonly used in Android apps?*
- **RQ3 (Acquiring and releasing points):** *At what program points are wake locks often acquired and released in Android apps?*

- **RQ4 (Critical computation):** *What computational tasks are often protected by wake locks in Android apps?*
- **RQ5 (Wake lock misuses):** *Are there common wake lock misuses in Android apps? What issues can they cause?*

RQ1–4 are designed to achieve our first goal. RQ5 is designed to achieve our second goal. By studying them, we obtained several useful findings. We give two examples here. First, we found that although in theory wake locks can be used to protect any computation from being disrupted by device sleeping, the usage of wake locks in practice is often closely associated with a small number of computational tasks. Second, by manually analyzing the bug reports and source code revisions of the 31 open-source apps, we identified 56 real wake lock issues. By categorizing them according to their root causes, we observed eight common types of wake lock misuses. Out of the eight types, existing work studied only two of them and proposed detection techniques for only one of them. Such findings not only provide guidance to developers when they need to use wake locks, but also facilitate future research on developing useful techniques for avoiding, detecting, debugging, and fixing wake lock issues. To summarize, our work has two major contributions:

- We conducted an empirical study of a large number of real-world Android apps by leveraging program analysis and statistical analysis techniques to understand how wake locks are used by developers in practice. To the best of our knowledge, we are the first to do so.
- We collected 56 real wake lock issues in open-source Android apps. By categorizing them, we observed eight types of common wake lock misuses. These findings can directly facilitate future research such as automated bug detection.

The remainder of this paper is organized as follows. Section 2 briefs Android app basics. Section 3 describes the methodology of our empirical study. Section 4 presents the study results. Section 5 discusses the threats to our study and feedback from developers. Section 6 reviews related work and finally Section 7 concludes this paper.

2. BACKGROUND

Android is a Linux-based mobile operating system [2]. Apps running on Android platforms are normally written in Java and compiled to Dalvik bytecode, which are then encapsulated into Android app package files (i.e., APK files) for distribution and installation [1].

2.1 App Components and Event Handling

App components. An Android app typically comprises four types of components: *activities*, *services*, *broadcast receivers*, and *content providers* [1]. Activities are the only components that have graphical user interfaces (GUIs) for interacting with users. Services are components that run at background for performing long-running operations. Broadcast receivers define how an app responds to system-wide broadcast messages. Content providers manage shared app data for other components or apps to query or modify.

Event handling. Android apps are event-driven programs. An app's logic is normally implemented in a set of *event handlers*: callback methods that will be invoked by the Android framework when their corresponding events occur. To provide rich user experience, the Android platform defines thousands of such handlers to process various events. Here, we briefly introduce two important types [39]: (1) app component lifecycle event handlers, and (2) GUI event handlers. The former type of handlers processes an app component's lifecycle events (e.g., creation, pausing, and termination). For instance, an activity's `onCreate` handler will be invoked upon the activity's creation. The latter type of handlers processes user interaction events on an app's GUI. For example, the `onClick` handler of a GUI widget (e.g., button) will be invoked to handle user clicks on the widget.

Table 1: Different types of wake locks and their impact on system power consumption

Wake lock type	CPU	Screen	Keyboard backlight
Partial	On	Off	Off
Screen dim	On	Dim	Off
Screen bright	On	Bright	Off
Full	On	Bright	Bright
Proximity screen off	Screen off when proximity sensor activates		

2.2 Wake Lock Mechanism

Wake locks enable developers to explicitly control the power state of an Android device. To use a wake lock, developers need to declare the `android.permission.WAKE_LOCK` permission in their app's manifest file. After obtaining the permission, they need to create a `PowerManager.WakeLock` instance and specify the type of the wake lock. Table 1 lists all five wake locks types supported by the Android framework.[1] Each type has a different wake level and thus different effects on the system's power consumption. For instance, a full wake lock will ensure that: (1) the device CPU keeps running, and (2) the screen and keyboard backlight are on and at full brightness. After creating wake lock instances, developers can then invoke corresponding APIs to acquire and release wake locks (see Figure 8 for example). Once acquired, a wake lock will have a long lasting effect on the whole system until it is released or the specified timeout expires. When acquring wake locks, developers can also set certain flags. For example, setting the `ON_AFTER_RELEASE` flag will cause the device screen to remain on for a while after the corresponding wake lock is released. Due to wake locks' direct effect on device hardware state, developers should carefully use them to avoid undesirable consequences (e.g., energy waste).

3. METHODOLOGY

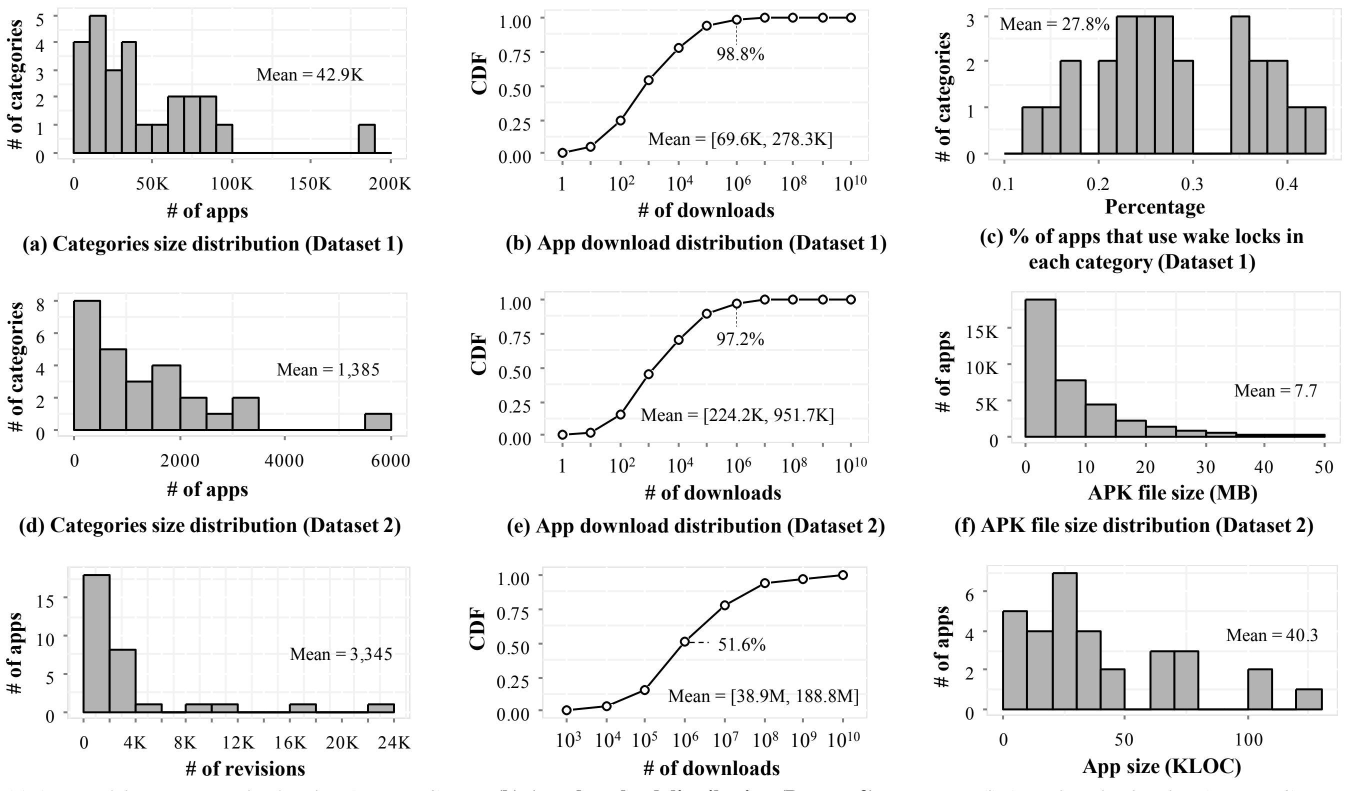
This section presents our methodology for the empirical study. We start by introducing our datasets and then discuss how we analyze them to answer our research questions.

3.1 Dataset Collection

For our study, we collected the following three datasets:

Dataset 1: Basic information of commercial apps. We first collected the basic information of 1,117,195 commercial Android apps from Google Play store using a web crawler [7]. Specifically, for each app, we collected its: (1) app ID, (2) category, (3) number of downloads, and (4) declared permissions. Figure 1(a)–(c) give some statistics of the dataset. The apps in the dataset cover all 26 major app categories defined by Google Play store [5] and each category contains thousands of apps (Figure 1(a)). Many of the

[1]Latest Android versions deprecate screen bright, screen dim and full wake locks and suggest developers to set the `KEEP_SCREEN_ON` flag when defining activities. The OS will then keep device screen on when the activities are visible to users. Although this may ease programming, for finer-grained control, developers still often use wake locks as we will see in Section 4.1.



(a) Categories size distribution (Dataset 1)

(b) App download distribution (Dataset 1)

(c) % of apps that use wake locks in each category (Dataset 1)

(d) Categories size distribution (Dataset 2)

(e) App download distribution (Dataset 2)

(f) APK file size distribution (Dataset 2)

(g) App revision number distribution (Dataset 3)

(h) App download distribution (Dataset 3)

(i) App size distribution (Dataset 3)

Figure 1: Statistics of our datasets ("CDF" stands for Cumulative Distribution Function)

Table 2: Top 5 categories of apps that commonly use wake locks

App category	% of apps that declare wake lock permission
News & Magazines	42.6%
Music & Audio	41.4%
Media & Video	38.6%
Communication	38.0%
Business	37.9%

apps are popularly downloaded from market. Figure 1(b) gives the distribution of downloads of these apps. As we can see, nearly half of the apps have achieved one thousand downloads and 1.2% of the apps have over one million downloads. On average, each app has received more than 69.6 thousand downloads.[2] Besides, by checking permissions, we found that in many categories, a large percentage of apps declare the permission to use wake locks (Figure 1(c)). Table 2 lists the top five categories and it shows that wake locks are popularly used in real-world Android apps.

Dataset 2: Binaries of commercial apps. Our research questions RQ1–4 require analyzing the code of Android apps to answer. Therefore, we also need to collect the binaries (APK files) of those Android apps that use wake locks. For this purpose, we first obtained a list of apps that declare the wake lock permission from our first dataset. This list contains 303,877 candidate apps. We then randomly downloaded the binaries of 36,020 apps from Google Play store using the `APKLeecher` tool [4]. We were not able to download more apps for this study because the process was very time-consuming, and our download got blocked many times by Google servers. Collecting the 36,020 apps took us four months using tens of PCs and servers with different IPs. The downloaded apps cover all categories and each category contains a sufficient number of apps for our study (Figure 1(d)). These apps are popular on market: more than half of them have achieved thousands of downloads and 1,008 (2.8%) of them have been downloaded millions of times (Figure 1(e)). We also give the size distribution of these APK files in Figure 1(f). On average, each APK file takes 7.7 MB disk space.

Dataset 3: Source repositories of open-source apps. Answering RQ5 (wake lock misuses) requires studying the bug reports and source code revisions of Android apps. Such data are typically only available in open-source apps. Therefore, we also need to collect open-source Android apps that use wake locks. To find suitable subjects, we searched four major open-source software hosting platforms: GitHub [12], Google Code [13], SourceForge [21], and Mozilla repositories [16]. We aimed to look for those apps that satisfy three requirements: (1) the app has over 1,000 downloads (popularity), (2) the app has a public issue tracking system (traceability), and (3) the app has at least hundreds of code revisions (maintainability). In the end, we collected 31 open-source Android apps. Figure 1(g)–(i) gives the statistics of these apps (see Table 4 for examples). As we can see, the apps are very popularly downloaded from market: 15 (48.4%) of them have achieved millions of downloads. They are well-maintained, containing hundreds or thousands of code revisions. Besides, they are large-scale. On average, each of them contains 40.3 thousand lines of code.

3.2 Analysis Algorithms

Program analysis. To answer RQ1–4, we analyzed the first two datasets. Figure 2 illustrates how we analyzed the

[2]The user download is not an exact number as Google Play store only provides a range (e.g., 500 - 1,000).

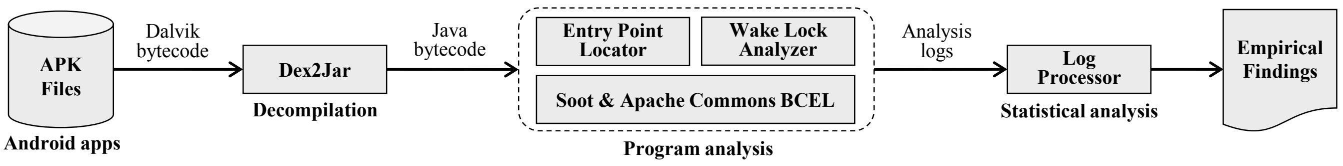


Figure 2: Methodology for analyzing APK files

App component	*lock*	api_1	api_2	api_3	api_4
C_1	1	0	1	1	1
C_2	0	1	0	1	0
C_3	0	1	1	0	0

Figure 3: An example bit matrix

APK files. We first decompiled the Dalvik bytecode in each APK file to Java bytecode using `Dex2Jar` [9]. We then analyzed the Java bytecode for each app using a static analysis tool we implemented on top of the `Soot` program analysis framework [20] and Apache Byte Code Engineering Library (`BCEL`) [3]. The tool first performs class hierarchy analysis to identify all app component classes (e.g., those extending the `Activity` class) in an app and locates the set of event handlers defined in each app component (e.g., `onCreate` handlers of activities and `onClick` handlers of GUI widgets). These event handlers will serve as the entry points of our analysis. Then, our tool constructs a call graph for each entry point and traversed the graph in a depth-first manner to check whether wake lock acquiring and releasing API calls can be transitively reached. If yes, we consider that the corresponding app component would use wake locks. We understand that such call graphs may not be precise and complete. However, statically constructing precise and complete call graphs for Java programs is a well-known challenge due to the language features such as dynamic method dispatching [23]. Addressing this challenge is out of our study scope, but still, to ensure the precision of our analysis results, we would remove the obvious imprecise call graph edges caused by conservative resolution of virtual calls while traversing the call graphs (see Section 5 for more discussion). During the analysis, our tool logs the following information for later statistical analysis: (1) the set of app components that use wake locks, (2) the type of the used wake locks, (3) all entry points where wake locks are acquired and released, and (4) the set of APIs invoked by each app component.

Statistical analysis. With the data logged during program analysis, we can analyze them to answer RQ1–4. Answering RQ1–3 requires only straightforward statistical analysis and we do not further elaborate. Answering RQ4 (critical computation) requires us to investigate the computational tasks that are performed by our collected apps. To do so, we first analyzed the APIs invoked by the apps in our second dataset, since API usage typically reflects the computational semantics of an app [42]. Specifically, our goal is to identify those critical APIs that are frequently invoked in app components that use wake locks (*locking app component* for short), but not frequently invoked in app components that do not use wake locks (*non-locking app components* for short). These APIs are very likely invoked by the apps to conduct the critical computations. Before we explain how to identify such APIs, we first formally define our problem.

Suppose that we analyze a set $\mathcal{S}$ of apps. Each app $A \in \mathcal{S}$ can contain a set of n app components $comp(A) = \{C_1, C_2, \ldots, C_n\}$, where $n \geq 1$. Then, the whole set of app components to analyze is:

$$comp(\mathcal{S}) = \bigcup_{A \in \mathcal{S}} comp(A).$$

With our above discussed program analysis, for each app component $C \in comp(\mathcal{S})$, we will know whether it uses any wake lock or not, and the set of APIs it invokes. Let us use $\mathcal{API}$ to denote the ordered set of all possible APIs. Then, after analyzing the apps in $\mathcal{S}$, we can encode the results as a bit matrix $\mathcal{M}$ of size $|comp(\mathcal{S})| \times (|\mathcal{API}| + 1)$. Each row of the matrix is a bit vector that encodes the analysis result for a corresponding app component. The first bit of the vector indicates whether the app component uses any wake lock or not ("1" represents "yes" and "0" represents "no"). The remaining $|\mathcal{API}|$ bits indicate whether corresponding APIs are invoked by the app component or not. To ease understanding, we give an example of bit matrix in Figure 3. This example involves three app components. The bit vector in the first row means that the app component C_1 uses wake locks and invokes three different APIs, namely, api_2, api_3 and api_4. The other rows can be interpreted similarly.

Now after the formulation, our problem can be reduced to the classic *term-weighting* problem in the natural language processing area with the following mappings [26]:

- The set of analyzed app components $comp(\mathcal{S})$ can be considered as a *corpus*;
- Each app component $C \in comp(\mathcal{S})$ can be considered as a *document*;
- Each API invokded by C can be considered as a *term* in the document that C represents;
- The set of locking app components can be considered as the *positive category* in the corpus;
- the set of non-locking app components can be considered as the *negative category* in the corpus.

With this reduction, we can adapt term-weighting techniques to identify those APIs that are frequently invoked by locking app components but not frequently invoked by non-locking app components. To do so, we choose to apply the widely-used *Relevance Frequency* approach [26]. Formally, for each API *api*, we count the following:

- a: the number of locking app components that invoke *api*;
- b: the number of locking app components that do not invoke *api*;
- c: the number of non-locking app components that invoke *api*;
- d: the number of non-locking app components that do not inovke *api*.

Then we define the *importance score* of *api* by Equation 1. The $rf(api)$ computes the relevance frequency score of *api* and its definition in Equation 3 follows the standard one [26]. The filtering function $freqFilter(api)$ defined by Equation 2 is to avoid assigning a high importance score to those APIs that do not frequently occur in locking app components and very rarely or never occur in non-locking app components (when a is a very small number compared to b, but $a \gg c$, the relevance frequency score will be exceptionally high). Such APIs are of less interest to us and assigning them high scores will waste our manual effort since we will study the APIs with high importance scores to answer RQ4.

$$importance(api) = freqFilter(api) \times rf(api) \qquad (1)$$

$$freqFilter(api) = \begin{cases} 1, & \text{if } a/(a+b) \geq 0.05 \\ 0, & \text{otherwise} \end{cases} \qquad (2)$$

$$rf(api) = \log\left(2 + \frac{a}{\max(1, c)}\right) \qquad (3)$$

In addition to API usage analysis, we also analyzed the permissions of the apps in our first two datasets to answer RQ4. We conducted permission analysis because Android apps must declare corresponding permissions to perform certain operations that may impact other apps, the operating system, or users (e.g., network access) [1]. Therefore, the declared permissions are also good indicators of the computations performed by an app. To identify the permissions that are frequently declared in apps that use wake locks, but not frequently declared in apps that do not use wake lock, we use the same algorithm as in API usage analysis. The only difference is: API usage analysis was conducted at app component level, while permission analysis was conducted at app level. Due to page limit, we skip more details.

Search-assisted manual analysis. To answer RQ5 (wake lock misuses), we manually studied the bug reports and code revisions of the 31 open-source apps, aiming to find wake lock related issues. These apps in total contain thousands of code revisions and bug reports. To save manual effort, we implemented a tool to search the apps' code repositories and bug tracking systems for: (1) those interesting bug reports that contain certain keywords, and (2) those interesting code revisions whose commit log or code diff contain certain keywords. The keywords include: *wake*, *wakelock*, *power*, *powermanager*, *acquire*, and *release*. After search, 1,157 bug reports and 1,558 code revisions meet our requirement. We then carefully studied them to answer RQ5.

4. STUDY RESULTS

We ran our analysis tasks on a Linux server with 16 cores of Intel Xeon CPU @2.10GHz and 192GB RAM. The majority of CPU time (~380 hours) was spent on the program analysis of the 36,020 apps with APK files. In total, these apps defined 715,823 activities, 146,002 services, 351,762 broadcast receivers, 15,274 content providers, and registered 3,468,567 GUI event listeners. Our tool successfully analyzed 35,327 APKs and found 45,818 app components using wake locks. The remaining 693 (1.9%) APKs failed to be analyzed because `Dex2Jar` or `Soot` crashed when processing them. In this section, we discuss some major results.[3]

4.1 Seven Facts about Wake Lock Usage

For RQ1, we analyzed apps in each category and studied how many of them acquire wake locks in each type of app components. Figure 4 gives the results of three categories as examples ("Overall" represents all 35,327 apps). From the results, we can make two major observations.

Most apps acquire wake locks in broadcast receivers. Overall, 65.3% of our analyzed apps acquire wake

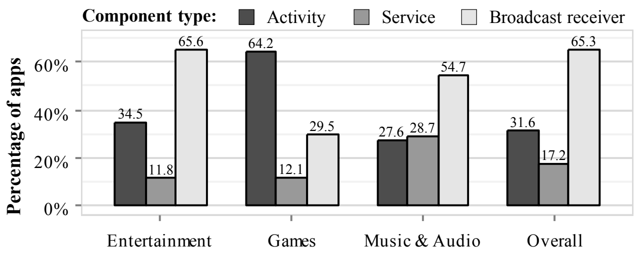


Figure 4: The percentage of apps that use wake locks in each type of app components

locks in broadcast receivers. As a comparison, only 31.6% and 17.2% acquire wake locks in activities and services, and no apps acquire wake locks in content providers. These percentages do not add up to 100% as some apps acquire wake locks in multiple components. This finding reveals a common practice in real-world app design: many apps actively listen to certain messages (e.g., servers' push notifications in email apps) and will keep device awake to handle received ones so as to notify users the results (e.g., new emails).

Interaction-intensive and presentation apps often acquire wake locks in activities. Although for 23 of the 26 categories, most apps acquire wake locks in broadcast receivers (e.g., Entertainment and Music & Audio apps in Figure 4), in the following three categories, most apps acquire wake locks in activities: Games, Books & References, and Family. Games and Family apps (e.g., kid education apps) heavily interact with users; Books & References apps are designed for presenting reading materials. These apps often need to keep device screen on when users launch them (e.g., starting their main activities). This may explain why they often acquire wake locks in activities.

For RQ2, we analyzed the types of wake locks used in different categories of apps. As we discussed earlier, different wake locks have different effects on devices' power state. So what types of wake locks do developers commonly use in reality? We discuss three findings below.

Full wake locks are commonly used in activities, while partial wake locks are commonly used in services and broadcast receivers. Overall, as shown in Figure 5(a), 44.2% of the wake locks used in activities are full wake locks and over 82% wake locks used in services and broadcast receivers are partial wake locks. This finding is intuitive. Activities contain GUIs for user interactions and thus they often need to keep device screen on (full and screen bright/dim wake locks can satisfy this requirement). On the other hand, services and broadcast receivers perform computational tasks at background, and a partial wake lock that keeps device CPU on is sufficient to prevent the on-going computation from being disrupted by device sleeping.

Proximity screen off wake locks are rarely used in practice. Compared with the other types, proximity screen off wake locks are rarely used in our analyzed apps. We only observed their presence in a few categories of apps (e.g., Communication apps). There are three primary reasons for this phenomenon. First, the corresponding APIs were not introduced until Android 5.0. Second, to use this type of wake locks, the devices need to have proximity sensors, which are not always available on Android devices [1]. Third, the wake locks are designed for a specific purpose: when the proximity sensor detects that some objects are nearby, the device screen will be turned off immediately to avoid accidental screen taps that can adversely change the running status of some apps (e.g., a call may be ended when

[3]For all results, please refer to our project website at `http://sccpu2.cse.ust.hk/wl_empirical/` (ID:lxct, Password:icse).

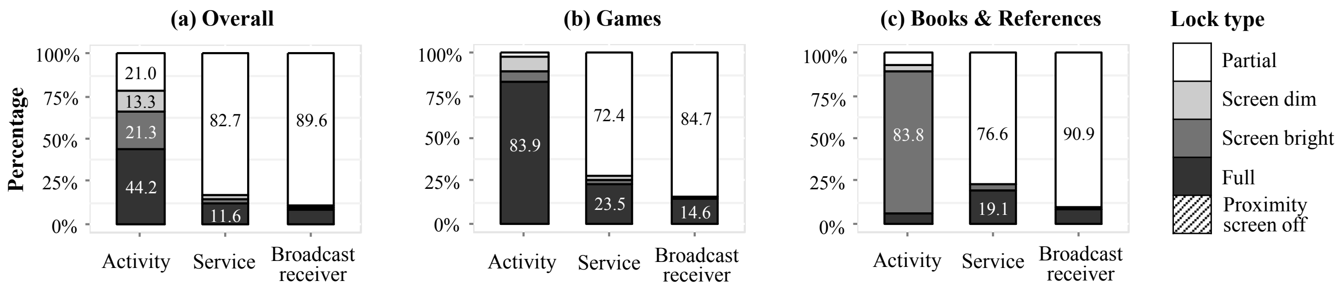


Figure 5: Distribution of lock types w.r.t different types of app components

Event handlers implemented in activity components (H_1 – H_9) and in service components (H_{10} – H_{15}):

H_1	`Activity.onResume()`	H_6	`Activity.onDestory()`	H_{11}	`Service.onStart()`
H_2	`Activity.onCreate()`	H_7	`Activity.onStart()`	H_{12}	`Service.onStartCommand()`
H_3	`Activity.onPause()`	H_8	`Activity.onStop()`	H_{13}	`Service.onCreate()`
H_4	`View$OnClickListener.onClick()`	H_9	`DialogInterface$OnCancelListener.onCancel()`	H_{14}	`Service.onDestroy()`
H_5	`Activity.onWindowFocusChanged()`	H_{10}	`IntentService.onHandleIntent()`	H_{15}	`DreamService.onDreamingStarted()`

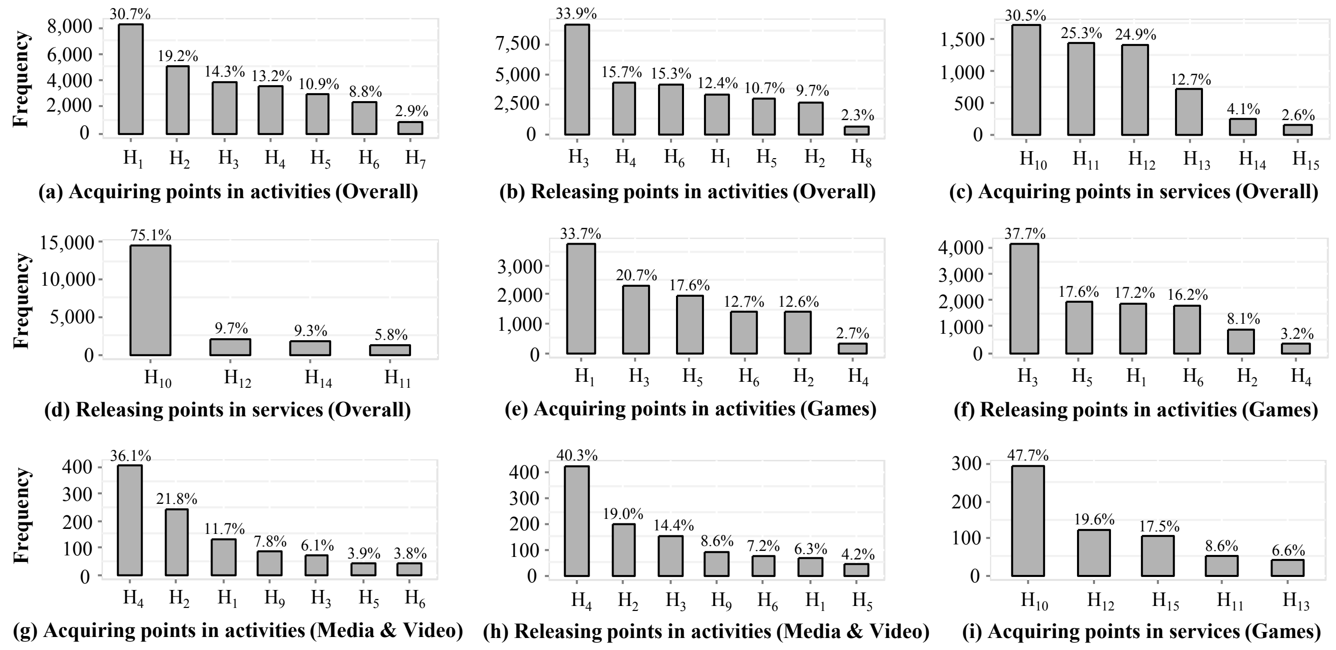


Figure 6: Common wake lock acquiring and releasing points in activity and service components

users' cheek accidentally touches phone screens).

Interaction-intensive apps often use full wake locks. Presentation apps often use screen bright wake locks. Albeit most app categories exhibit similar results, we observed two exceptions: (1) full wake locks are very frequently used by the activity components in Games and Family apps ((see Figure 5(b)) for example), and (2) screen bright wake locks are very frequently used by the activity components in Books & References apps (see Figure 5(c)). As we discussed previously, this is largely due to the nature of these apps.

For RQ3, we analyzed which event handlers commonly acquire and release wake locks in different categories of apps. Figure 6 gives the results for several example categories. We discuss two major observations.

Wake locks are commonly acquired and released in several major lifecycle event handlers. Developers may acquire and release wake locks at different program points. For example, in our analyzed apps, the wake locks used in activity components can be acquired in 102 different event handlers (including many GUI event handlers). However, despite the diversity, in most cases, wake locks are only acquired and released in a few event handlers:

- Overall, in activities, wake locks are mostly acquired in `onResume` handlers (Figure 6(a)), which will be invoked by the system right before the activities are ready for user interaction, and released in `onPause` handlers (Figure 6(b)), which will be invoked right after the activities lose user focus. *This is a good practice for avoiding energy waste as many apps do not need to keep device awake for computation when they are switched to background by users.*
- In services, wake locks are mostly acquired and released in `onHandleIntent` handlers (see Figure 6(c) and (d)), which are invoked by the system to perform long running background computations in worker threads.
- For broadcast receivers, the Android framework only defines one event handler `onReceive`. Therefore, wake locks are all acquired and released there. Such wake locks are typically used to ensure that the concerned apps can smoothly finish handling certain important broadcast messages.

Wake lock acquiring and releasing points can be category-specific. Similar to RQ1–2, we also observe catgory-specific results for RQ3. For instance, we found that the activities of gaming apps do not commonly acquire and

Table 3: Computational tasks that are commonly protected by wake locks

Computational tasks	# related permissions	Permission example	# related APIs	API example
Networking	10	Receive data from Internet	112	`java.net.DatagramSocket.connect()`
App UI rendering	N/A	N/A	110	`android.graphics.Canvas.drawARGB()`
File I/O	3	Access USB storage file system	105	`android.os.Environment.getExternalStorageDirectory()`
Inter-component communication	1	Send sticky broadcast	68	`android.content.ContextWrapper.sendBroadcast()`
Security & privacy	3	Use accounts on the device	46	`javax.crypto.SecretKeyFactory.generateSecret()`
Media & audio	1	Record audio	32	`android.media.AudioTrack.play()`
System/user data accessing	11	Modify your contacts	25	`android.webkit.CookieSyncManager.run()`
Database transactions	N/A	N/A	24	`android.database.sqlite.SQLiteDatabase.update()`
Camera operations	2	Take pictures and videos	17	`android.hardware.Camera.setParameters()`
Sensing operations	4	Precise location	16	`android.location.Location.getLatitude()`
System setting	10	Modify system settings	12	`android.provider.Settings$System.putInt()`
User notification	1	Control vibration	12	`android.app.NotificationManager.notify()`
System-level operations	10	Close other apps	10	`android.os.Process.killProcess()`
Telephony services	4	Directly call phone numbers	9	`android.telephony.TelephonyManager.listen()`
Bluetooth/NFC communication	3	Pair with Bluetooth devices	5	`android.bluetooth.BluetoothAdapter.getDefaultAdapter()`

Notes: (1) "N/A" means that the corresponding computational tasks do not require special permissions. (2) The table is sorted according to # related APIs.

release wake locks in `onClick` handlers (see Figure 6(e) and (f)), as compared to other categories. On the contrary, `onClick` handlers are the most common wake lock acquiring and releasing points in the following categories of apps: Communication, Media & Video, Productivity, and Social (see Figure 6(g) and (h) for examples). Again, the phenomenon is largely due to the nature of these apps. Games do not often use standard GUI widgets (e.g., buttons). Instead, they often rely on graphical libraries (e.g., `OpenGL ES`) to render interfaces and thus do not often implement GUI event handlers like `onClick`. On the other hand, apps in categories like Media & Video often conduct long running tasks (e.g., video playing) after users click certain GUI widgets (e.g., the "play" button) and it is necessary to acquire wake locks in the corresponding `onClick` handlers. Another interesting example is that only in gaming apps, the `onDreamingStarted` handlers are common wake lock acquiring points (see Figure 6(i)). We sampled some of such apps and found that they all provide screen saving service. The `onDreamingStarted` handlers are invoked by the system to render screen savers and thus need to keep device screen on.

4.2 Reasons for Staying Awake

Our research question RQ4 aims to identify the critical computational tasks that are frequently protected by wake locks. As mentioned in Section 3, to identify such tasks, we performed permission analysis on the 1,117,195 apps in our first dataset and API usage analysis on the 35,327 apps in our second dataset. Specifically, for each category of apps, our tool computed the importance score for each permission declared and each API used in the apps, and ranked the permissions and APIs according to the computed scores (a higher score leads to a higher rank). Overall, we observe that these apps declare 236 different permissions and use 37,241 different APIs, 91.5% of which are official Android and Java APIs. After the analyses and ranking, we then manually examined the top 10% permissions (around 10 to 20) and top 1% APIs (around 100 to 200) for each of the 26 app categories to answer RQ4. Now we present our observations.

Theoretically, wake locks can be used to prevent devices from falling asleep during any kind of computation, which could be app-specific. However, by the analysis of a large number of apps, we observe that **developers often only use wake locks to protect a small number of computational tasks**. Particularly, our manual examination identified 63 permissions and 962 APIs that frequently occur in apps/app components that use wake locks, but not frequently occur in other apps/app components. We then categorized these permissions and APIs according to their documentations and design purposes [1]. For example, APIs in `android.database` and `java.sql` packages are designed for database transactions and we categorize them into one major category. After such categorization, we observed that these permissions and APIs are mainly designed for 15 types of computational tasks. Table 3 lists the categorization results. For each type of computational task, the table reports the number of related permissions and APIs, and provides examples to ease understanding.[4] We can see from the table that **most of the listed computational tasks can bring users observable or perceptible benefits**. Take network communication for example. Many apps frequently communicate with remote servers and fetch data to present to users (e.g., an social app). These tasks typically should not be disrupted by device sleeping when users are using the apps and expecting to see certain updates (e.g., messages from friends). Hence, wake locks are needed in such scenarios. Another typical example is security & privacy. We found that a large percentage of apps (e.g., 92.7% Finance apps) frequently encrypt and decrypt certain program data (e.g., those related to user privacy) for security concerns. Such tasks should also be protected by wake locks as any disruption can cause serious consequences to users.

4.3 Common Wake Lock Misuses

For research question RQ5, we manually investigated the 1,157 bug reports and 1,558 code revisions found by keyword search (see Section 3). This process is labor-intensive, taking us several months to finish. During the process, we found 56 real issues caused by wake lock misuses in the investigated bug reports and code revisions. Other bug reports and code revisions are irrelevant to wake lock issues, but were accidentally included because they contain our searched keywords. We carefully studied these 56 issues and categorized them after understanding their root causes. By the categorization, we observed eight types of common wake lock issues. We present the results in Table 4. For each type, the table: (1)

[4]We failed to categorize 359 of the 962 APIs into any major categories because they are general-purpose (e.g., HashMap APIs).

Table 4: Common types of wake lock issues found in open-source Android apps

Root cause	# issues	Example issues					
		App name	App downloads	Bug report ID	Issue fixing revision	Consequence	
Wake lock leakage	12	MyTracks [17]	10M - 50M	N/A	1349	Energy waste	
Unnecessary wakeup	10	Tomahawk [22]	5K - 10K	N/A	883d210525	Energy waste	
Premature lock releasing	9	ConnectBot [6]	1M - 5M	37	540c693d2c	Crash	
Multiple lock acquisition	8	CSipSimple [8]	1M - 5M	152	153	Crash	
Inappropriate lock type	8	SipDroid [19]	1M - 5M	268	533	Instability	
		Osmand [18]	1M - 5M	582	4d1c97fe7768	Energy waste	
Problematic timeout setting	3	K-9 Mail [15]	5M - 10M	170/175	299	Instability	
Inappropriate flags	2	FBReader [10]	10M - 50M	N/A	f28986383f	Energy waste	
Permission errors	2	Firefox [16]	100M - 500M	703661	be42fae64e	Crash	

Notes: For some issues, we failed to locate the associated bug reports (the issues may not be documented) and the corresponding cells are marked as "N/A".

```
1.      public class ExportAllAsyncTask extends AsyncTask {
2.       public ExportAllAsyncTask() {
3.          wakeLock = MyTrackUtils.acquireWakeLock();
4.        }
5.        protected Boolean doInBackground(Void... params){
6.          Cursor cursor = null;
7.          try {
8.            cursor = MyTrackUtils.getTracksCursorFromDB();
9.            for(int i = 0; i < cusor.getCount(); i++) {
10.             if(isCancelled()) break;
11.             exportAndPublishProgress(cursor, i);
12.           }
13.         } finally {
14.           if(cursor != null) cursor.close();
15.   +       if(wakeLock.isHeld()) wakeLock.release();
16.         }...
17.       }
18.       protected void onPostExecute(Boolean result){
19.   -     if(wakeLock.isHeld()) wakeLock.release();
20.       }
21.     }
```

Figure 7: Wake lock leakage in MyTracks

```
1.      public class MusicActivity extends Activity implements...{
2.        public void onCreate() {
3.          startService(new Intent(PlaybackService.class);
4.        }
5.        public void onTouch() {
6.          PlaybackService.getInstance().playPause();
7.        }
8.        public void onDestroy() {
9.          stopService(new Intent(PlaybackService.class));
10.       }
11.     }
12.     public class PlaybackService extends Service {
13.       public void onStart(Intent i) {
14.         wakeLock = newWakeLock(PowerManager.PARTIAL_WAKE_LOCK);
15.         wakeLock.acquire(); //then start music playing
16.       }
17.       public void playPause() {
18.         if(mediaPlayer.isPlaying()) {
19.           mediaPlayer.pause();
20.   +       if(wakeLock.isHeld()) wakeLock.release();
21.         } else {
22.   +       if(!wakeLock.isHeld()) wakeLock.acquire();
23.           mediaPlayer.start();
24.         }
25.       }
26.       public void onDestroy() {
27.         if(wakeLock.isHeld()) wakeLock.release();
28.       }
29.     }
```

Figure 8: Unnecessary wakeup in Tomahawk

briefly summarizes the root cause (Column 1), (2) lists the number of similar issues we found in the open-source apps (Column 2), and (3) provides a typical example (Columns 3–7). We now discuss these issues in more detail below.

Wake lock leakage. The most common issues are *wake lock leakages*. As we mentioned earlier, wake locks should be properly released after use. However, ensuring wake locks to be released on all program paths for event-driven programs like Android apps is a non-trivial task. Developers often make mistakes. Figure 7 gives an example wake lock leakage issue in MyTracks [17], a popular app for recording users' tracks when they exercise outdoors. The app defines an long-running task `ExportAllAsyncTask` to export recorded tracks to external storage (e.g., an SD card). When the task starts, it acquires a wake lock (Line 3). Then it runs in a worker thread to read data from database, writes them to the external storage, and notifies users the exporting progress (Lines 5–17). When the job is done, the Android system will invoke the `onPostExecute` handler, which will release the wake lock (Line 19). This process works fine in many cases. Unfortunately, developers forgot to handle the cases where users cancel the exporting task before it finishes. In such cases, `onPostExecute` will not be invoked after the `doInBackground` method returns. Instead, another handler `onCancel` will be invoked. Then, the wake lock will not be released properly. The consequence is that the device cannot go asleep, causing significant energy waste. Later, developers realized this issue and moved the lock releasing operation to the `doInBackground` method (Line 15).

Unnecessary wakeup. The second common type is *unnecessary wakeup*. we observe that in many apps, wake locks are correctly acquired and released, but the lock acquiring and releasing time is not appropriate. They either acquire wake locks too early or release them too late, causing the device to stay awake unnecessarily. To ease understanding, we discuss a real issue in Tomahawk [22], a music player app. Figure 8 gives the simplified code snippet. When users select an album, Tomahawk's `MusicActivity` will start a service `PlaybackService` to play music in background (Lines 2–4). When the service is launched, it acquires a partial wake lock, sets up the media player, and starts music playing (Lines 13–16). Users can pause or resume music playing by tapping the device screen (Lines 5–7 and 17–25). When users exit the app, `MusicActivity` and `PlaybackService` will be destroyed and the wake lock will be released accordingly (Lines 8–10 and 26–28). This is functionally correct and the music can be played smoothly in practice. However, since the wake lock is used to keep the device awake for music playing, why should it be held when the music player is paused? Holding unnecessary wake locks can lead to serious energy waste. Developers later fixed the issue by releasing the wake lock when music playing is paused (Line 20) and re-acquire it when users resume music playing (Line 22).

Premature lock releasing. The third common type is *premature lock releasing*. These issues occur when a wake lock is released before being acquired and they can cause app crashes (e.g., ConnectBot issue 37 [6]). In our studied apps, we frequently observed such issues. One major reason is that Android apps can have complex control flows due to the event-driven programming paradigm. If developers do not fully understand the lifecycle of different app components (e.g., temporal relationships between event handlers), they may mistakenly place a wake lock releasing operation in an event handler that can be executed before another one that acquires the wake lock.

Multiple lock acquisitions. Wake locks by default are reference counted. Each acquiring operation on a wake lock increments its internal counter and each releasing operation decrements the counter. The Android OS only releases a wake lock when its associated counter reaches 0 [1]. Due to this policy, developers should avoid *multiple lock acquisitions*. Otherwise, to release a wake lock requires an equivalent number of lock releasing operations. However, due to complex control flows, developers often make mistakes that cause a wake lock to be acquired multiple times. For example, in CSipSimple [8], a popular Internet call app, developers put the wake lock acquiring operation in a frequently invoked method. The consequence is that CSipSimple crashes after acquiring a wake lock too many times (issue 152), which exceeds the upper bound allowed by the system.

Inappropriate lock type. Before using wake locks, developers need to figure out which hardware needs to stay awake for the critical computation, and choose an appropriate type of wake lock for the purpose. Choosing inappropriate types of wake locks often causes troubles in practice. For example, in SipDroid [19], another popular Internet call app, developers used a partial wake lock for keeping the device CPU awake during Internet calls. However, on many devices, keeping CPU awake does not prevent WiFi NIC from entering Power Saving Polling mode, which will significantly reduce the network bandwidth. The consequence is that SipDroid's calling quality becomes unstable when device screen turns off (issue 268). To fix the issue, developers later used a screen dim wake lock to keep both device screen and CPU on when users are making phone calls. This is an example of mistakenly using a wake lock with a low wake level. We also observed cases where developers use wake locks whose wake levels are higher than necessary. For instance, when users use Osmand [18], a famous maps & navigation app, to record their trips during outdoor activities (e.g., cycling), the app will acquire a screen dim wake lock for location sensing and recording. However, keeping screen on in such scenarios is unnecessary and will waste a significant amount of battery energy (Osmand issue 582). Developers later realized the issue after receiving many user complaints and replaced the screen dim wake lock with a partial wake lock.

Problematic timeout setting. When acquiring wake locks, developers can set a timeout. Such wake locks will be automatically released after the given timeout. Setting an appropriate timeout (enough for the critical computation to complete, but not too long) seems to be an easy job. However, in practice, developers can make bad estimations. For example, in K-9 Mail [15], an email client with millions of users, developers used a wake lock that would timeout after 30 seconds to protect the email checking process. They thought the duration was long enough for the checking to complete. Unfortunately, due to various reasons (e.g., slow network conditions), many users complained that they often fail to receive email notifications and this issue is annoyingly intermittent. The developers later found the root cause. They reset the timeout to 10 minutes and commented:

"This should guarantee that the syncing never stalls just because a single attempt exceeds the wake lock timeout."

Inappropriate flags. We mentioned in Section 2 that developers can set certain pre-defined flags when acquiring wake locks. When they do so, they need to be careful as setting *inappropriate flags* can cause unexpected consequences. For example, the developers of FBReader [10], an eBook reading app, found that setting the `ON_AFTER_RELEASE` flag when using a screen bright wake lock could cause serious energy waste on some users' devices. This is because, with the flag set, some Android system variants (i.e., those customized by device manufacturers) can keep the device screen on at full brightness for quite a long while after the wake lock is released. They later removed the flag to fix the issue.

Permission errors. Using wake locks requires an app to declare the `android.permission.WAKE_LOCK` permission. Forgetting to do so will lead to security violations. This is a well-documented policy [1], but developers still make mistakes. For example, one version of Firefox did not declare the permission properly and users found that the app would crash because of this issue (Firefox issue 703661 [11]).

The above issues are recurring ones in our studied apps. We also observed two other types of issues that only occurred once. One is the instability issue caused by concurrent wake lock acquiring and releasing in K-9 Mail. Developers fixed the issue by putting wake lock operations in synchronized blocks (revision 1698 [15]). The other is the duplicate wake lock issue in CSipSimple. Developer mistakenly made two app components acquire the same type of wake locks, but one is already sufficient. They later fixed the issue by removing one wake lock (revision 1633 [8]).

Potential research opportunities. We discussed eight types of wake lock misuses that commonly caused functional and non-functional issues in practice. We observed that the first two types have been explored by existing work. There already exist several techniques for wake lock leakage detection [24, 30, 35]. Nonetheless, the majority of our identified issues have not been well-studied by our community. Therefore, we believe that in future, some research effort can be spent on designing effective and efficient techniques to help developers catch such wake lock misuses. We discuss two possibilities. First, we think that formulating certain criteria or heuristic rules (e.g., by leveraging our findings for RQ4) to help developers reason about the necessity of using wake locks at different app states is a meaningful direction. Some dynamic monitoring techniques can be proposed. Second, it is also a feasible direction to design testing/analysis techniques that can systematically traverse different program paths to automatically locate multiple lock acquisition and premature lock releasing issues. For this direction, how to address the path explosion problem is worth exploring.

5. DISCUSSIONS

Threats to validity. The validity of our study results may be subject to several threats. The first is the representativeness of our analyzed Android apps. To minimize the threat, we randomly downloaded the latest version of a large number of commercial apps from Google Play store and selected 31 popular and large-scale open-source apps from four major software hosting platforms. So we believe our findings can generalize to many real-world Android apps. The second threat is the precision of our statically constructed call graphs. We understand that imprecise call graphs may lead to imprecise findings. Therefore, before analyzing the whole dataset, we did a pilot study on 50 randomly sampled apps, 30 commercial and 20 open-source. Indeed, we found the call graphs generated by Soot can contain infeasible edges. The primary reason is that Soot cannot precisely resolve the targets of virtual calls. Its default algorithm would conservatively consider all possible callees based on class hierarchy.[5] This led to some counter-intuitive analysis results. For instance, we found many activities in a commercial app UzVoIP [14] would release wake locks in onCreate handlers, which are the starting points of the activities' lifetime. After checking the decompiled code, we realized that these handlers transitively invoke the run method of a Runnable object. Unfortunately, the app contains another class that implements the Runnable interface and its run method invokes a wake lock releasing API. To address the problem, we adopted a simple strategy in our analysis: during call graph traversal, if a visited method has multiple callees with the same method signature, we will not further visit such callees. This solution worked quite well in our study and we will show later that our empirical findings are confirmed by real-world developers. The third threat is our manual investigation of bug reports and code revisions of open-source apps. We understand that this manual process can be error-prone. To reduce the threat, we tried our best to cross-validate the results and we will also release them for public access.

Developer feedback. Our work aims to understand how Android developers use wake locks. It would be interesting to see how real-world developers perceive our findings. For this purpose, we conducted a survey. We randomly selected 20,000 apps from our first dataset and invited their developers to complete a survey form that mainly asks three open questions: (1) *in which scenarios they think an app needs to use wake locks*, (2) *what mistakes they have encountered when using wake locks*, and (3) *whether the tools in Android SDK are helpful in debugging wake lock issues*. For the first two questions, we provided some of our findings as examples to help them formulate answers. For the third question, we listed four most commonly used debugging tools in Android SDK (e.g., DDMS) for their references. Till now, there have been 278 developers completed our survey and 124 of them often use wake locks. Based on the 124 experienced developers' answers, we made some observations. First, our findings indeed reflect developers' common practices. The participants generally confirmed that they often use wake locks to protect the critical computations we identified. Besides, some of them also encountered issues caused by our observed wake lock misuses. One participant mentioned that using multiple wake locks of different types once crashed his app, but we did not observe similar issues in our study. Second, tools provided by Android SDK may not be helpful in debugging wake lock issues: 31% participants think that SDK tools are not helpful in debugging crashing bugs, and 48% think that the tools are not helpful in debugging energy bugs. The participants also commented on what tool support they would like to have. We give an example below:

> *"What one would need is a tool which realistically simulates the passage of time and typical daily user activity. Wake lock bugs usually cause trouble when an app is used over a long period of time on a phone where many other apps are competing for the system's resources."*

Finally, we asked the participants whether they would like to use a tool, which we were developing, to help diagnose wake lock misuses. Encouragingly, 68 of the 124 participants expressed their interest and left contact information for us to follow up. This suggests that developers have limited access to tools that can help them properly use wake locks.

[5]Other strategies can construct more precise call graphs, but suffer from scalability problems and thus are not suitable for analyzing a large number of apps [38].

6. RELATED WORK

Our paper relates to a large body of existing work on improving energy efficiency of Android apps. In this section, we discuss some representative pieces of work in recent years.

There are several studies that specifically focused on wake locks issues. Pathak et al. conducted the first study of energy waste bugs in smartphone apps, and proposed to use reaching-definition dataflow analysis algorithms to detect no-sleep energy bugs caused by wake lock leakage [35]. Later, along this direction, researchers proposed many other useful techniques [24, 30, 36, 37]. For example, Guo et al. reduced wake lock leakage to the traditional resource leakage problem and proposed to use classic resource leakage detection techniques to detect wake lock issues [24]. Wang et al. designed a technique to detect wake lock issues and repair them at runtime to minimize energy waste [37]. Nonetheless, these pieces of work mostly focused on wake lock leakage issues. As our empirical findings suggest, there are many other types of wake lock misuses in practice. It is desirable to design effective techniques to help detect and fix them.

There are also studies that can help improve the energy efficiency of Android apps from other angles [28]. For example, vLens [27], eLens [25], eProf [34], PowerTutor [41] can help developers estimate the energy consumption of their apps by different models to identify energy hotspots for optimization. ADEL [40] and GreenDroid [29] can help locate energy bugs in Android apps caused by ineffective use of high energy cost program data (e.g., network data). Besides, the most recent work AlarmScope [33] proposed a technique to reduce non-critical alarm-induced wakeups in Android apps to minimize energy waste. These techniques are mostly designed for app developers. Researchers also proposed end user-oriented techniques. For example, eDoctor [31] can correlate system and user events to energy-heavy execution phases and help end users troubleshoot abnormal battery drains and suggest repairs. Carat [32] shares the same goal as eDoctor, but adopts a collaborative and big data-driven approach. It collects runtime data from a large community of smartphones for inferring energy usage models so as to provide users actionable advices on improving smartphone battery life.

7. CONCLUSION

In this paper, we conducted a large-scale empirical study on 1.1 million commercial and 31 open-source Android apps to understand how real-world developers use wake locks in their apps. Our study discovered some common practices of developers and identified eight types of wake lock misuses that frequently cause functional and non-functional issues in Android apps. These findings can guide developers, especially inexperience ones, to appropriately use wake locks and support research on designing effective techniques to avoid, detect, debug and fix wake lock issues. In future, we plan to design a profiling-based technique to help developers reason about the optimal placement of wake lock operations to avoid energy waste caused by unnecessary wakeup. We hope that our work together with related work can help improve the energy efficiency of Android apps, which can potentially benefit millions of users around the world.

8. REFERENCES

[1] Android developers. https://developer.android.com/.

[2] Android official website. http://www.android.com/.

[3] Apache Commons BCEL. http://commons.apache.org/proper/commons-bcel/.

[4] APKLeecher. http://apkleecher.com/.

[5] AppBrain statistics. http://www.appbrain.com/.

[6] ConnectBot source repository. https://code.google.com/p/connectbot/.

[7] Crawler4j. https://code.google.com/p/crawler4j.

[8] CSipSimple source repository. https://code.google.com/p/csipsimple/.

[9] Dex2Jar. https://code.google.com/p/dex2jar.

[10] FBReader source repository. https://github.com/geometer/FBReaderJ.

[11] Firefox issue tracker. https://bugzilla.mozilla.org/.

[12] GitHub. https://github.com/.

[13] Google Code. https://code.google.com/.

[14] Google Play store. https://play.google.com/store.

[15] k-9 Mail source repository. https://code.google.com/p/k9mail/.

[16] Mozilla repositories. https://mxr.mozilla.org/.

[17] MyTracks source repository. https://code.google.com/p/mytracks/.

[18] Osmand source repository. https://code.google.com/p/osmand/.

[19] SipDroid source repository. https://code.google.com/p/sipdroid/.

[20] Soot: a Java Program Optimization Framework. http://sable.github.io/soot/.

[21] SourceForge. http://sourceforge.net/.

[22] Tomahawk source repository. https://github.com/tomahawk-player/tomahawk.

[23] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, Atlanta, Georgia, USA, 1997.

[24] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in Android applications. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, pages 389–398, Nov 2013.

[25] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 92–101, San Francisco, CA, USA, 2013.

[26] M. Lan, C. Tan, J. Su, and Y. Lu. Supervised and Traditional Term Weighting Methods for Automatic Text Categorization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(4):721–735, April 2009.

[27] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating Source Line Level Energy Information for Android Applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, Lugano, Switzerland, 2013.

[28] Y. Liu, C. Xu, and S. Cheung. Diagnosing energy efficiency and performance for mobile internetware applications. *IEEE Software*, 32(1):67–75, Jan 2015.

[29] Y. Liu, C. Xu, and S. C. Cheung. Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. In *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications*, PerCom '13, pages 2–10, San Diego, CA, USA, March 2013.

[30] Y. Liu, C. Xu, S. C. Cheung, and J. Lu. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering*, 40(9):911–940, Sept 2014.

[31] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 57–70, Lombard, IL, 2013.

[32] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, Roma, Italy, 2013.

[33] S. Park, D. Kim, and H. Cha. Reducing Energy Consumption of Alarm-induced Wake-ups on Android Smartphones. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, pages 33–38, Santa Fe, New Mexico, USA, 2015.

[34] A. Pathak, Y. C. Hu, and M. Zhang. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, Bern, Switzerland, 2012.

[35] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, 2012.

[36] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal. Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs. In *Proceedings of the 2012 Workshop on Power-Aware Computing and Systems*, Hollywood, CA, 2012. USENIX.

[37] X. Wang, X. Li, and W. Wen. Wlcleaner: Reducing energy waste caused by wakelock bugs at runtime. In *Proceedings of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing*, DASC '14, pages 429–434, Aug 2014.

[38] X. Xiao and C. Zhang. Geometric Encoding: Forging the High Performance Context Sensitive Points-to Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 188–198, Toronto, Ontario, Canada, 2011.

[39] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *Proceedings of the 37th International Conference on Software Engineering*,

ICSE '15, Florence, Italy, May 2015.

[40] 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 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 363–372, Tampere, Finland, 2012.

[41] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '10, pages 105–114, Oct 2010.

[42] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1105–1116, Scottsdale, Arizona, USA, 2014.