Understanding and Detecting Wake Lock Misuses for Android Applications

Yepang Liu[§] Chang Xu[‡] Shing-Chi Cheung[§] Valerio Terragni[§]

[§]Dept. of Comp. Science and Engineering, The Hong Kong Univ. of Science and Technology, Hong Kong, China
[‡]State Key Lab for Novel Software Tech. and Dept. of Comp. Sci. and Tech., Nanjing University, Nanjing, China
[§]{andrewust, scc*, vterragni}@cse.ust.hk, [‡]changxu@nju.edu.cn*

ABSTRACT

Wake locks are widely used in Android apps to protect critical computations from being disrupted by device sleeping. Inappropriate use of wake locks often seriously impacts user experience. However, little is known on how wake locks are used in real-world Android apps and the impact of their misuses. To bridge the gap, we conducted a large-scale empirical study on 44,736 commercial and 31 open-source Android apps. By automated program analysis and manual investigation, we observed (1) common program points where wake locks are acquired and released, (2) 13 types of critical computational tasks that are often protected by wake locks, and (3) eight patterns of wake lock misuses that commonly cause functional and non-functional issues, only three of which had been studied by existing work. Based on our findings, we designed a static analysis technique, Elite, to detect two most common patterns of wake lock misuses. Our experiments on real-world subjects showed that Elite is effective and can outperform two state-of-the-art techniques.

CCS Concepts

•Software and its engineering → Software testing and debugging; Software performance; •General and reference → Empirical studies; •Human-centered computing → Smartphones;

Keywords

Wake lock, critical computation, lock necessity analysis

1. INTRODUCTION

Nowadays, smartphones are equipped with powerful hardware components such as HD screen and GPS sensor to provide rich user experience. However, such components are big consumers of battery power. To prolong battery life, many smartphone platforms like 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 [58].

Albeit preserving energy, this aggressive sleeping policy may break the functionality of those 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 poor user experience. To address this problem, modern smartphone platforms allow apps to explicitly control when 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 a corresponding 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.

Wake locks are widely used in practice. We found that around 27.2% of apps on Google Play store [21] use wake locks for reliably providing certain functionalities (see Section 3). Despite the popularity, correctly programming wake locks is a non-trivial task. In order to avoid undesirable consequences, a conscientious developer should carefully think through the following questions before using wake locks:

1. *Do the benefits of using wake locks justify its energy cost (for reasoning about the necessity of using wake locks)?*
2. *Which hardware components need to stay awake (for choosing the correct type of wake lock)?*
3. *When should the hardware components 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 use wake locks in an undisciplined way. For example, our investigation of 31 popular open-source Android apps, which use wake locks, revealed that 19 (61.3%) of them have suffered from various functional and non-functional issues/bugs caused by wake lock misuses. These issues caused lots of user frustrations. Yet, existing work only studied a small fraction of them [38, 51, 58, 61]. Developers still lack guidance on how to appropriately use wake locks and have limited access to useful tools that can help locate their mistakes. To bridge the gap, we performed a large-scale empirical study on 44,736 commercial and 31 open-source Android apps, aiming to investigate three important research questions:

- **RQ1 (Acquiring and releasing points):** *At what program points are wake locks often acquired and released?*
- **RQ2 (Critical computation):** *What computational tasks are often protected by wake locks?*
- **RQ3 (Wake lock misuse patterns):** *Are there common patterns of wake lock misuses? What kind of issues can they cause?*

By automated program analysis of commercial apps and manual investigation of the bug reports and code revisions

[*] Corresponding authors

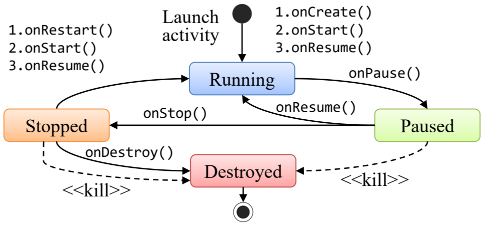


Figure 1: Lifecycle of an activity component

of open-source apps, we made several important observations. For example, we found that although in theory wake locks can be used to protect any computation from being disrupted by device sleeping, in practice, the usage of wake locks is often closely associated with a small number of computational tasks. We also identified 55 real wake lock issues from the 31 open-source apps. By studying their root causes, we observed eight common patterns of wake lock misuses, only three of which had been studied by existing work. Such findings can provide programming guidance to Android developers, and support follow-up research on developing techniques for detecting, debugging, and fixing wake lock issues.

Based on our empirical findings, we designed a static analysis technique, Elite, to detect two most common patterns of wake lock misuses. Unlike existing techniques [38, 61], Elite makes no assumption on where wake locks should be acquired and released, but automatically reasons about the necessity of using wake locks at various program points via dataflow analysis. We evaluated Elite using six real issues from real-world open-source subjects, and compared it with two existing techniques [38, 61]. Elite effectively located five of the six issues without generating any false alarm. As a comparison, the two existing techniques only located one real issue and 12 of their reported 13 warnings are spurious. To summarize, our work makes three major contributions:

- We conducted a large-scale empirical study to understand how Android developers use wake locks in practice. To the best of our knowledge, this study is the first of its type.
- We collected 55 real wake lock issues in 31 popular open-source Android apps [16]. By categorizing them, we observed eight common patterns of wake lock misuses.
- We designed and implemented a static analysis technique Elite to detect common wake lock misuses. Our evaluation of Elite on real-world subjects showed that Elite is effective and can outperform existing techniques.

Paper organization: Section 2 briefs Android app basics. Sections 3–4 describe our empirical study methodology and findings. Sections 5–6 presents and evaluates our Elite technique. Section 7 discusses threats to validity. Section 8 reviews related work and Section 9 concludes this paper.

2. BACKGROUND

Android is a Linux-based mobile OS [5]. Android apps are typically 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

An Android app typically comprises four types of components [1]: (1) *activities* contain graphical user interfaces (GUIs) for interacting with users; (2) *services* run at background for performing long-running operations; (3) *broadcast receivers* respond to system-wide broadcast messages; and (4) *content providers* manage shared app data for queries.

Table 1: Wake lock types and their impact

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		

*: These types are deprecated in latest Android versions, but still often used.

```
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
WakeLock wl = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "lockTag");
wl.acquire(); //acquire wake lock
//performing critical computation when the wake lock is held
wl.release(); //release the wake lock when critical computation completes
```

Figure 2: Example code for using wake locks

Android apps are event-driven. Their logic is normally implemented in a set of *event handlers*: callbacks that will be invoked by the Android OS when certain events occur. To provide rich user experience, the Android platform defines thousands of handlers to process various events, of which we introduce three major types [64]:

1) *Component lifecycle event handlers* process an app component's lifecycle events (e.g., creation, pausing, and termination). For example, Figure 1 gives the lifecycle of an activity. When the activity is created, the handlers `onCreate()`, `onStart()`, and `onResume()` will be invoked consecutively.

2) *GUI event handlers* process user interaction events on an app's GUI, which usually consists of standard widgets (e.g., buttons) and custom views [4]. For example, a button's `onClick()` handler will be invoked if the button gets clicked by user and its `onClickListener` is registered.

3) *System event handlers* process system-level events monitored by the Android OS such as incoming calls and sensor updates. These handlers (e.g., `onLocationChanged()`) will be invoked if the corresponding events occur and their listeners (e.g., `LocationListener`) are registered.

2.2 Wake Lock Mechanism

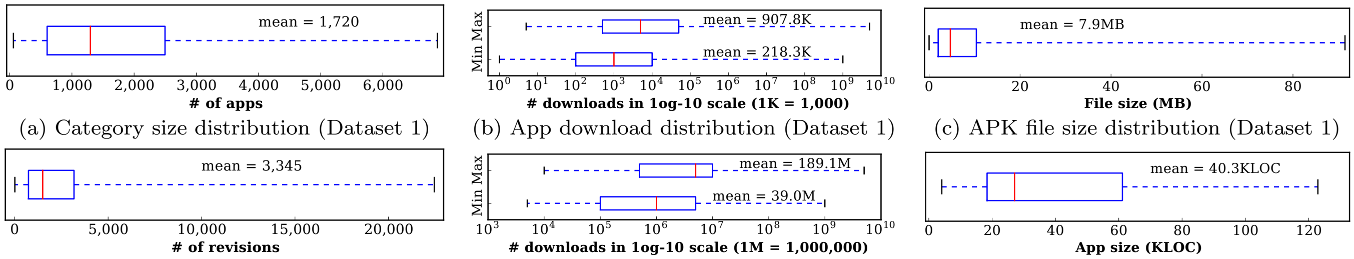
Wake locks enable developers to explicitly control the power state of an Android device. To use a wake lock, developers need to declare the `android.permission.WAKE_LOCK` permission in their app's manifest file [7, 36], create a `PowerManager.WakeLock` instance, and specify its type (see Figure 2). Table 1 lists the five types of wake locks supported by the Android framework. Each type has a different wake level and affects system power consumption differently. For instance, a full wake lock will keep device CPU running, and screen and keyboard backlight on at full brightness. After creating wake lock instances, developers can invoke certain APIs to acquire and release wake locks. Once acquired, a wake lock will have long lasting effects until it is released or the specified timeout expires. When acquiring 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 wake lock is released. Due to wake locks' direct effect on device hardware state, developers should carefully use them to avoid undesirable consequences.

3. EMPIRICAL STUDY METHODOLOGY

This section presents our datasets and how we analyze them to answer our research questions.

3.1 Dataset Collection

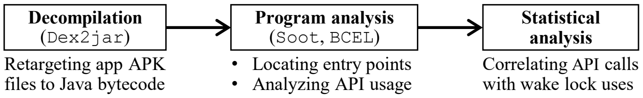
Dataset 1: Binaries of 44,736 Android apps. Answering RQ1–2 requires analyzing the code of Android apps that use wake locks. For this purpose, we collected the binaries (APK files) of 44,736 Android apps from Google Play



(a) Category size distribution (Dataset 1) (b) App download distribution (Dataset 1) (c) APK file size distribution (Dataset 1)

(d) Revision number distribution (Dataset 2) (e) App download distribution (Dataset 2) (f) App size distribution (Dataset 2)

Figure 3: Dataset statistics (Figures (b) and (e) have two boxplots because Google Play store provides a download range)


Decompilation (Dex2jar)

Retargeting app APK files to Java bytecode

Program analysis (Soot, BCEL)

- Locating entry points
- Analyzing API usage

Statistical analysis

Correlating API calls with wake lock uses

Figure 4: Methodology for analyzing APK files

ALGORITHM 1. Finding the set of dynamically registered GUI and system event *handlers* in an component class *c*

```
1. $worklist, handlers \leftarrow$ findAllLifeCycleHandlers($c$)
2. while $worklist$ $not$ $empty$ do
3.     $h \leftarrow$ dequeue($worklist$)
4.     $regSites \leftarrow$ findEventListenerRegistrationSites($h$)
5.     foreach $regSite \in regSites$ do
6.         $h_{new} \leftarrow$resolveEventHandler($regSite$)
7.         if $h_{new} \neq null$ and $h_{new} \notin handlers$ then
8.             add $h_{new}$ to $handlers$ and $worklist$
9. return $handlers$
```

store by the following process. First, we collected the basic information (app ID, category, and declared permissions) of 1,117,195 Android apps using a web crawler [13]. By permission analysis, we found that 303,877 (27.2%) of these apps declare permissions to use wake locks. We then proceeded to download these apps using `APKLeecher` [9]. In total, we tried 200,231 randomly selected apps and successfully obtained 44,736 of them. The downloading took 11 months (March 2015 to January 2016) with more than ten PCs and one server. Figure 3 gives the statistics of our downloaded apps. They cover all 26 app categories [10] and each category on average contains 1,720 apps (Figure 3(a)). They are popular on market: more than half of them have achieved thousands of downloads and 1,318 (2.9%) of them have been downloaded millions of times (Figure 3(b)). We also give the size distribution of the downloaded APK files in Figure 3(c). On average, each APK file takes 7.9 MB disk space.

Dataset 2: Bug reports and code repositories. Answering RQ3 requires studying the bug reports and code revisions of Android apps that use wake locks. Such data are typically only available for open-source apps. To find subjects for our study, we searched four major open-source software hosting sites: GitHub [19], Google Code [20], SourceForge [29], and Mozilla repositories [23]. We aimed to find those apps that have: (1) over 1,000 downloads (popular), (2) a public issue tracking system (traceable), and (3) over 100 code revisions (well-maintained). After manually checking more than 200 candidates, we identified 31 apps that use wake locks and satisfy the three requirements. Figure 3 gives the statistics of these apps. As we can see, they are popular on market: 15 (48.4%) of them have achieved millions of downloads (Figure 3(e)). They are also well-maintained, containing hundreds to thousands of code revisions (Figure 3(d)). Besides, they are large-scale. On average, each of them contains 40.3 thousand lines of code (Figure 3(f)).

3.2 Analysis Algorithms

Program analysis. To answer RQ1–2, we analyzed the 44,736 APK files. Figure 4 illustrates the overall process. We first decompiled each APK file to Java bytecode using `Dex2Jar` [15]. We then analyzed each app's Java bytecode using a static analysis tool we built on the `Soot` program analysis framework [28] and Apache Byte Code Engineering Library (`BCEL`) [8]. The analysis consists of two major steps:

Step 1: Locating analysis entry points. Our tool first performs class hierarchy analysis to identify all app component classes (e.g., those extending the `Activity` class) in an app. It then locates the set of callbacks defined in each app component, including lifecycle event handlers, GUI event handlers, and system event handlers. Lifecycle event handlers can be located by searching for the corresponding overwritten methods in an app component class. Finding GUI and system event handlers requires a more sophisticated searching algorithm because event handlers can freely register other event handlers. To find all dynamically registered handlers,[1] our tool adopts a fixed-point iterative searching strategy, which is illustrated by Algorithm 1. The high level idea is to iteratively extend the set of located event handlers, which initially only contains component lifecycle event handlers (Line 1), by adding new ones that are registered in latest located event handlers (Lines 2–8). To find the event handlers registered by a certain event handler h, our algorithm traverses the call graph of h, pinpoints all event listener registration sites (Line 4), and resolves the type of each listener to locate its associated event handler (Line 6). All located event handlers will serve as the entry points in later analysis. It is worth mentioning that our tool also locates other entry points such as the callbacks defined in custom views [4] (e.g., `onDraw()`) and asynchronous tasks [3] (e.g., `doInBackground()`). Their locating algorithms are essentially similar to the algorithms explained here.

Step 2: Analyzing API calls. Then, for each app component, our tool traverses the call graph of each entry point in a depth-first manner to check whether wake lock acquiring and releasing API calls can be transitively reached. If yes, we consider the app component would use wake locks. We understand that the call graphs constructed by `Soot` may not be precise and complete. However, statically constructing precise and complete call graphs for Java programs is challenging due to the language features such as dynamic method dispatching and reflection [37]. Addressing this challenge is out of our scope, but still, to ensure the precision of our results, our analysis would not visit obviously imprecise call graph edges caused by conservative resolution of virtual calls (see Section 7). During the analysis, our tool

[1] GUI event handlers statically declared in an app's layout configuration files can be located by XML parsing.

logs the following information for later statistical analysis: (1) the set of app components that use wake locks, (2) the set of entry points where wake locks are acquired and released, and (3) the set of APIs invoked by each component. Note that for app components that use wake locks, we only log the APIs that can be invoked after the wake lock is acquired and before the wake lock is released.

Statistical analysis. With the data logged during program analysis, we can analyze them to answer RQ1–2. Answering RQ1 only requires straightforward statistical analysis and we do not further elaborate. Answering RQ2 requires investigating the computational tasks that are performed by our collected apps. To do so, we analyzed the APIs invoked by the apps in our dataset, since API usage typically reflects the computational semantics of an app [67]. Our goal is to identify those critical APIs that are commonly invoked by app components that use wake locks (*locking app components* for short), but not commonly invoked by app components that do not use wake locks (*non-locking app components*). 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 of apps $\mathcal{A}$. Each app $a \in \mathcal{A}$ can contain a set of n app components $C(a) = \{c_1, c_2, \ldots, c_n\}$, where $n \geq 1$. Then, the whole set of app components to analyze is $C(\mathcal{A}) = \bigcup_{a \in \mathcal{A}} C(a)$. With program analysis, for each analyzed app component $c \in C(\mathcal{A})$, we can know whether it uses wake locks or not, and the set of APIs it invokes. Let us use $\mathcal{API}$ to denote the ordered set of all concerned APIs. Then, after analyzing the apps in $\mathcal{A}$, we can encode the results as a bit matrix $\mathcal{M}$ of size $|C(\mathcal{A})| \times (1 + |\mathcal{API}|)$. 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 wake locks or not. The remaining $|\mathcal{API}|$ bits indicate whether corresponding APIs are invoked by the app component or not. With such formulation, we can reduce our problem to the classic *term-weighting* problem in the natural language processing area with the following mappings [42]:

The set of analyzed app components $C(\mathcal{A})$ can be considered as a *corpus*. Each app component $c \in C(\mathcal{A})$ can be considered as a *document*. Each API invoked 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 and the set of non-locking app components can be considered as the *negative category*.

With the reduction, we can adapt term-weighting techniques to identify the APIs that are commonly invoked by locking app components but not commonly invoked by non-locking app components. To do so, we applied the widely-used *Relevance Frequency* approach [42]. Formally, for each $api \in \mathcal{API}$, we count the following ("#" = "the number of"):

- α: # locking app components that invoke *api*;
- β: # locking app components that do not invoke *api*;
- γ: # non-locking app components that invoke *api*.

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 [42]. 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 α is very small compared to β, but $\alpha \gg \gamma$, the *rf* score will be exceptionally high). Assigning high scores to such uninteresting APIs will waste our effort as we will study the APIs with high importance scores to answer RQ2.

Table 2: Acquiring and releasing program points

(1) Results for activity components

Acquiring point	Pct.	Releasing point	Pct.
`onResume()`	30.5%	`onPause()`	35.4%
`onCreate()`	19.2%	`onDestroy()`	15.8%
`onPause()`	14.2%	`onResume()`	13.0%
`onWindowFocusChanged()`	10.8%	`onWindowFocusChanged()`	11.2%
`onDestory()`	8.8%	`onCreate()`	10.2%
Other 365 callbacks	16.5%	Other 389 callbacks	14.4%

(2) Results for service components

Acquiring point	Pct.	Releasing point	Pct.
`onHandleIntent()`	26.5%	`onHandleIntent()`	68.2%
`onStartCommand()`	21.7%	`onStartCommand()`	13.6%
`onStart()`	21.2%	`onDestroy()`	8.6%
`onCreate()`	12.3%	`onStart()`	5.2%
`onMessage()`	3.8%	`onCreate()`	1.1%
Other 192 callbacks	14.5%	Other 188 callbacks	3.3%

(3) Results for broadcast receiver components*

Acquiring point	Pct.	Releasing point	Pct.
`onReceive()`	98.4%	`onReceive()`	93.8%
Other 37 callbacks	1.6%	Other 26 callbacks	6.2%

*: They only have one major lifecycle event handler `onReceive()`.

$$importance(api) = freqFilter(api) \times rf(api) \tag{1}$$

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

$$rf(api) = \log\left(2 + \frac{\alpha}{\max(1, \gamma)}\right) \tag{3}$$

Search-assisted manual analysis. To answer RQ3, we manually studied the bug reports and code revisions of the 31 open-source apps, aiming to find wake lock misuse issues. These apps contain thousands of code revisions and bug reports. To save manual effort, we wrote a Python script 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 RQ3.

4. EMPIRICAL OBSERVATIONS

We ran the analysis tasks on a Linux server with 16 cores of Intel Xeon CPU @2.10GHz and 192GB RAM. The majority of CPU time was spent on decompiling the 44,736 APK files (~206 hours) and the program analysis (~307 hours). In total, these apps defined 893,374 activities, 185,672 services, 447,302 broadcast receivers, 19,370 content providers, and 5,190,728 GUI/system event listeners. We successfully analyzed 43,888 (98.1%) APKs and found 52,816 app components that use wake locks. The remaining 848 APKs failed to be analyzed because `Dex2Jar` or `Soot` crashed when processing them. In this section, we discuss our major findings.

4.1 RQ1: Acquiring and Releasing Points

First, most apps acquire wake locks in broadcast receivers.[2] Overall, 65.2% of our analyzed apps acquire wake locks in broadcast receivers. For activities and ser-

[2] Due to Android fragmentation [6, 54], screen dim/bright and full wake locks still account for 12.8% and 21.9% of all used ones.

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

Computational task	# related APIs	API examples	# related permissions	Permission example
Networking & Communications	103	`java.net.DatagramSocket.connect()`	9	Receive data from Internet
Logging & file I/O	100	`android.os.Environment.getExternalStorageDirectory()`	4	Access USB storage file system
Asynchronous computation	82	`java.lang.Thread.start()/android.os.AsyncTask.execute()`	0	N/A
UI & graphics rendering	79	`android.opengl.GLSurfaceView.setRenderer()`	1	Draw over other apps
Inter-component communication	48	`android.content.ContextWrapper.sendBroadcast()`	1	Send sticky broadcast
Data management & sharing	40	`android.database.sqlite.SQLiteDatabase.query()`	0	N/A
System-level operations	34	`android.os.Process.killProcess()`	5	Close other apps
Media & audio	33	`android.media.AudioTrack.write()`	1	Record audio
Security & privacy	28	`javax.crypto.SecretKeyFactory.generateSecret()`	3	Use accounts on the device
Sensing operations	16	`android.location.LocationManager.requestLocationUpdates()`	4	Precise location
Alarm & notification	10	`android.app.NotificationManager.notify()`	1	Control vibration
System setting	6	`android.provider.Settings$System.putInt()`	2	Modify system settings
Telephony services	2	`android.telephony.TelephonyManager.listen()`	1	Directly call phone numbers

Notes: "N/A" means that the corresponding computational tasks do not require special permissions.

vices, the percentages are 30.8% and 17.3%, respectively. We did not find any app that acquires wake locks in content providers. Note that these percentages do not add up to 100% because some apps acquire wake locks in multiple components. This finding suggests that in practice wake locks are often used by apps when they process broadcast messages, which are usually sent upon the occurrence of important events (e.g., servers' push notifications).

We further analyzed each type of app components to investigate in which callbacks developers often acquire and release wake locks. Table 2 summarizes the results. We can observe that **wake locks are commonly acquired and released in major lifecycle event handlers**. For example, in activities, wake locks are mostly acquired in `onResume()` handlers, which are invoked by Android OS when the activities are ready for user interaction, and released in `onPause()` handlers, which are invoked after the activities lose user focus. *This reveals developers' common 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.* Besides the major lifecycle event handlers, we can also observe that **developers may acquire and release wake locks at various other program points depending on the needs of their apps**. For instance, our analyzed activity components acquire wake locks in 370 different callbacks. This is out of our expectation. Previous techniques [38, 61] for analyzing wake lock misuses often assume that wake locks are acquired when an app component launches and should be released at a set of program exit points. This finding suggests that such assumptions may not hold in many cases and effective techniques should not rely on pre-defined rules. Instead, they should consider each app case by case by analyzing its semantics.

4.2 RQ2: Critical Computation

RQ2 aims to identify the critical computational tasks that are frequently protected by wake locks. To identify such tasks, we performed API usage analysis on the 43,888 successfully analyzed apps. For each category of apps, our tool computed the importance score of each invoked API and ranked the APIs according to their scores (a higher score leads to a higher rank). Overall, these apps use 34,957 different APIs, 87.6% of which are official Android and Java APIs. After the analyses and ranking, we manually examined the top 1% APIs commonly used by each of the 26 app categories to answer RQ2. 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 analyzing a large number of apps, we observe that **developers often only use wake locks to protect several types of computational tasks**. Particularly, we identified 807 APIs that are commonly invoked by locking app components, but not commonly invoked by other app components. We then categorized these APIs according to their design purposes [1]. For example, APIs in `android.database` and `java.sql` packages are designed for data management and we would categorize them into the same category. After categorization, we observed that these APIs are mainly designed for 13 types of computational tasks and many of them require the Android OS to grant certain permissions to run [7, 36]. Table 3 summarizes our categorization results. For each type of computational task, the table reports the number of related APIs and their required system permissions, and provides examples to ease understanding.[3] We can see from the table that **these computational tasks, many of which are run asynchronously and could be long running (e.g., location sensing and media playing), can bring users observable or perceptible benefits**. Take networking & communications for example. Many apps frequently fetch data from remote servers and present them to users. These tasks typically should not be disrupted by device sleeping when users are using the apps and expecting to see certain updates. Hence, wake locks are needed in such scenarios. Another typical example is security & privacy. We found that a large percentage of apps (e.g., 84.5% 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. This finding can provide wake lock usage guidance to Android developers, and also facilitate the designing of useful techniques for detecting wake lock misuses (see Section 5).

4.3 RQ3: Wake Lock Misuse Patterns

For RQ3, we manually investigated the 1,157 bug reports and 1,558 code revisions found by keyword search. After investigation, we found that 55 bug reports and code revisions are related to wake lock misuses. The other bug reports and code revisions are irrelevant, but were accidentally included because they contain our searched keywords. We then carefully studied these 55 issues and categorized them after understanding their root causes. By the categorization, we observed eight common patterns of wake lock misuses (covering 53 issues and affecting 18 apps), which are listed in Table 4. For each pattern, the table lists the

[3] We failed to categorize 226 of the 807 APIs because they are general-purpose (e.g., HashMap APIs).

Table 4: Common patterns of wake lock misuses found in open-source Android apps

Misuse pattern	# issues	# affected apps	Example issues				
			App name	Downloads	Bug report ID	Issue fixing revision	Consequence
Unnecessary wakeup*	11	7	TomaHawk [30]	5K - 10K	N/A	883d210525	Energy waste
Wake lock leakage*	10	7	MyTracks [24]	10M - 50M	N/A	17ece1cd75	Energy waste
Premature lock releasing	9	7	ConnectBot [12]	1M - 5M	37	540c693d2c	Crash
Multiple lock acquisition	8	3	CSipSimple [14]	1M - 5M	152	153	Crash
Inappropriate lock type	8	3	SipDroid [27]	1M - 5M	268	533	Instability
			Osmand [26]	1M - 5M	582	4d1c97fe7768	Energy waste
Problematic timeout setting	3	2	K-9 Mail [22]	5M - 10M	170/175	299	Instability
Inappropriate flags	2	2	FBReader [17]	10M - 50M	N/A	f28986383f	Energy waste
Permission errors*	2	2	Firefox [23]	100M - 500M	703661	be42fae64e	Crash

Notes: (1) 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 MusicActivity extends Activity implements…{
 2.      public void onCreate() {//start PlaybackService…}
 3.      public void onTouch() {
 4.        PlaybackService.getInstance().playPause();…}
 5.      public void onDestroy() {//stop PlaybackService…}}
 6.    public class PlaybackService extends Service {
 7.      public void onCreate() {…
 8.  -     wakeLock.acquire(); //partial wake lock
 9.  -     mediaPlayer.start();…}
10.      public void playPause() {
11.        if(mediaPlayer.isPlaying()) {
12.          mediaPlayer.pause();
13.  +       if(wakeLock.isHeld()) wakeLock.release();
14.        } else {
15.  +       if(!wakeLock.isHeld()) wakeLock.acquire();
16.          mediaPlayer.start();}}
17.      public void onDestroy() {
18.        if(wakeLock.isHeld()) wakeLock.release();
19.        if(mediaPlayer.isPlaying()) mediaPlayer.stop();…}}
```

Figure 5: Unnecessary wakeup in TomaHawk

number of issues we found, the number of affected apps, and provides a typical example. We noticed that only three of the eight patterns (marked with "*" in Table 4) have been studied by existing work [38, 61, 51, 36]. The remaining five patterns that concern 30 issues are previously-unknown. We now discuss each pattern in detail.

Unnecessary wakeup is the most common pattern. We observe that in many apps, wake locks are correctly acquired and released on all program paths, 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 [30], a music player app. Figure 5 gives the simplified code snippet. When users select an album, TomaHawk's `MusicActivity` will start the `PlaybackService` to play music at background (Line 2). When the service is launched, it acquires a partial wake lock, sets up the media player, and starts music playing (Lines 7–9). Users can pause or resume music playing by tapping the device screen (Lines 3–4 and 10–16). When users exit the app, `MusicActivity` and `PlaybackService` will be destroyed and the wake lock will be released accordingly (Lines 5 and 17–19). 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 13) and re-acquiring it when users resume music playing (Line 15).

Wake lock leakage is the second common pattern. 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 Android apps is a non-trivial task. Figure 6 gives an example wake lock leakage in MyTracks [24], a popular app for recording users' tracks when they exercise outdoors. The app defines a 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, write them to the external storage, and notify users the exporting progress (Lines 4–13). When the job is done, the Android OS will invoke the `onPostExecute()` callback, which will release the wake lock (Line 15). This process works fine in many cases. Unfortunately, developers forgot to handle the case where users cancel the exporting task before it finishes. In such a case, `onPostExecute()` will not be invoked after `doInBackground()` returns. Instead, another callback `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 `doInBackground()` (Line 13).

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

Figure 6: Wake lock leakage in MyTracks

Premature lock releasing is the third common pattern. It occurs when a wake lock is released before being acquired and can cause app crashes (e.g., ConnectBot issue 37 [12]). In our studied apps, we often observed such issues. One major reason is the complex control flows of Android apps due to the event-driven programming paradigm. If developers do not fully understand the lifecycle of different app components (e.g., temporal relations among callbacks), they may mistakenly release a wake lock in a callback 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 zero [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 [14], a popular Internet call app, developers put the wake lock acquiring operation in a frequently invoked callback. The consequence is that CSipSimple crashes after acquiring the wake lock too many times (issue 152), which exceeds the limit allowed by the OS.

ALGORITHM 2. Detecting wake lock misuses in an *app*

```
1.  foreach $component\ c \in app$ do
2.      if useWakeLock($c$) then
3.          $componentsUsingWakeLocks$.add($c$)
4.          summarizeTopLevelMethods($c$)
5.          inferTemporalConstraintsBetweenTopLevelMethods($c$)
6.  $warnings \leftarrow \varnothing$
7.  foreach $c \in componentsUsingWakeLocks$ do
8.      $seqs \leftarrow$ generateAllValidTopLevelMethodCallSeqs($c, length$)
9.      foreach $seq \in seqs$ do
10.         foreach $cp \in$ checkPoints($seq$) do
11.             $warnings$.add(analyzeLockNecessity($seq, cp$))
12. return aggregateByRootCause($warnings$)
```

Inappropriate lock type. Before using wake locks, developers should figure out which hardware needs to stay awake for the critical computation, and choose an appropriate type of wake lock. Inappropriate lock types often cause trouble in practice. For example, in SipDroid [27], 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 Save Polling mode [59], 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 [26], 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 estimation. For example, in K-9 Mail [22], an email client with millions of users, developers used a wake lock that would get timeout after 30 seconds to protect the email checking process. They thought that it 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 [17], 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 some 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 [18]).

The above issues recur throughout our studied apps. We also observed two other 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 [22]). The other is the duplicate wake lock issue in CSipSimple. Developers mistakenly made two app components acquire two different wake locks of the same type, but one is already sufficient. They later fixed the issue by removing one wake lock (revision 1633 [14]).

5. DETECTING WAKE LOCK MISUSES

Based on our empirical findings, we propose a static analysis technique ELITE (wake lock necessity analyzer) to detect the two most common patterns of wake lock misuses: unnecessary wakeup and wake lock leakage.

5.1 Algorithm Overview

The input of ELITE is an Android app's APK file. The output is a report of detected wake lock misuse issues. To ease issue diagnosis, ELITE also provides the method call sequences leading to each detected issue and the program points where wake locks should be released.

Algorithm 2 gives an overview of ELITE. To detect wake lock misuses in an Android app, ELITE explores different executions of each app component that uses wake locks to locate the problematic program points, where wake locks are not needed but acquired, by performing an interprocedural analysis (Lines 7–11). In Android apps, an app component's execution can be represented as a sequence of calls to the component's *top level methods* (or entry points). These methods include lifecycle/GUI/system event handlers (e.g., `onCreate()` in Figure 5), callbacks of custom views and asynchronous tasks (e.g., `doInBackground()` in Figure 6), and non-callback methods that are exposed for other components to invoke (e.g., `playPause()` in Figure 5). At runtime, top level methods of an app component are invoked by the Android OS or other components to handle various events. For event handling, top level methods may invoke other methods. Therefore, to emulate the executions of an app component c, ELITE identifies the top level methods of c and generates valid sequences of calls to such methods (Line 8 and Section 5.3). Then, for issue detection, ELITE analyzes each sequence and locates program points, where c unnecessarily holds wake locks (Lines 9–11 and Section 5.4). To improve analysis efficiency, before generating call sequences to interesting top level methods, ELITE first summarizes the methods' potential runtime behavior by an interprocedural analysis (Line 4 and Section 5.2) so that the summaries can be reused when analyzing method call sequences.

To make ELITE's analysis effective, we need to address two technical challenges. First, the execution of top level

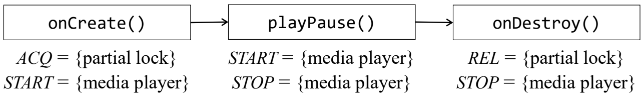


Figure 7: An example method call sequence

methods follow implicit orders prescribed by the Android platform. It is a non-trivial task to generate valid method call sequences. Second, there are no well-defined criteria to judge whether it is appropriate for an app to use wake locks at certain program points. To address the first challenge, ELITE infers temporal constraints from an app's code to model the execution orders of top level methods (Section 5.3). To address the second challenge, ELITE leverages our empirical findings and considers that if an app's computation can bring users perceptible or observable benefits, the energy cost of using wake locks is justified (Section 5.4).

5.2 Summarizing Top Level Methods

The first step of ELITE's analysis is to identify top level methods in each app component c that uses wake locks. To identify top level methods that are callbacks, which include lifecycle/GUI/system event handlers and callbacks defined in custom views and asynchronous tasks, ELITE relies on the fixed-point iterative searching algorithm described in Section 3.2. To identify other non-callback top level methods that c exposes for other components to invoke, ELITE iterates over non-callback methods defined in c and looks for those that can be directly invoked by other components without going through c's other methods. For each identified top level method m, ELITE then summarizes the following potential runtime behavior (dataflow facts) of m:

- *ACQ:* the set of wake lock instances that may have been acquired after executing m;
- *REL:* the set of wake lock instances that may have been released after executing m;
- *START:* the set of asynchronous computational tasks that may have been started after executing m;
- *STOP:* the set of asynchronous computational tasks that may have been stopped after executing m.

ELITE tracks the asynchronous computational tasks started and stopped by each analyzed method because these tasks are likely long running and could be the reason that an app uses wake locks (Section 4.2). To obtain these summaries, ELITE performs an interprocedural reaching-definition styled *forward dataflow analysis* [31] on the control flow graphs (CFGs) of all identified top level methods to infer the dataflow facts. The dataflow equations are defined as follows (s represents a statement in a CFG):

$$in(s) = \bigcup_{p \in pred(s)} out(p) \tag{4}$$

$$out(s) = gen(s) \cup (in(s) - kill(s)) \tag{5}$$

$in(s)$ and $out(s)$ represent the set of dataflow facts before and after executing s, respectively. $gen(s)$ and $kill(s)$ denote the dataflow facts generated or killed by s. The definitions of $gen(s)$ and $kill(s)$ depend on analysis tasks. For instance, when inferring $START$ of a method m, if a statement s invokes a location listener registration API, $gen(s)$ will have one element representing that s may start a long term location sensing task, and $kill(s)$ will be an empty set. At the control flow confluence and method return points, we use the union operator to combine data flow facts, meaning that $in(s)$ contains all dataflow facts from the predecessors of s in the CFG (Equation 4). $out(s)$ is a union of the dataflow facts generated by s and the difference between the dataflow facts s inherits from its predecessors and those killed by s (Equation 5). With these equations defined, ELITE then leverages `Soot`'s dataflow analysis engine to summarize all top level methods in app components that use wake locks.

5.3 Generating Valid Method Call Sequences

Next, ELITE proceeds to generate valid sequences of top level method calls for each app component that uses wake locks. ELITE considers a sequence valid if the invocation orders of the top level methods do not violate these temporal constraints prescribed by the Android platform [1, 51]:

Policy for lifecycle event handlers. Each app component typically passes through several phases during its lifecycle. In each phase, the corresponding lifecycle event handler(s) defined in the component class would be invoked by the Android OS at a specific time (Figure 1). ELITE leverages such temporal constraints to decide the correct invocation order of lifecycle event handlers.

Policy for GUI/system event handlers. An app component may register multiple GUI/system event listeners. The event handler defined in each event listener l can only be invoked when: (1) the top level method that registers l has been invoked, and (2) the top level method that unregisters l has not been invoked since then. ELITE also considers such constraints during method call sequence generation.

Policy for non-callback top level methods. Non-callback top level methods defined in an app component class are usually exposed for other components to invoke. For example, in Figure 5, the `playPause()` method of `PlaybackService` can be invoked by the `onTouch()` callback of `MusicActivity`. Such methods can only be invoked after their declaring app components are set up (e.g., after the `onStartCommand()` handler of `PlaybackService` is invoked).

Besides the three major types, in our implementation, ELITE also considers other temporal constraints to deal with the callbacks defined in asynchronous tasks [3] and custom views [4]. ELITE infers all constraints via static analysis. It then leverages them to generate valid method call sequences, each of which consists of three parts: (1) method calls to start components, (2) method calls to interact with the components, and (3) method calls to destroy the components. To ease understanding, Figure 7 provides a method call sequence ELITE generates for `PlaybackService` of TomaHawk (Figure 5). The sequence contains three top level method calls: (1) the call to `onCreate()` starts the service,[4] (2) the call to `playPause()` handles the user's touch on the `MusicActivity`'s UI, and (3) the call to `onDestroy()` destroys the service. It is worth mentioning that the number of all possible method call sequences can be unbounded since users can interact with an app in infinite ways. For practical consideration, ELITE limits the number of top level method calls in each sequence to generate a finite set of sequences. Other than this limit, ELITE's sequence generation is exhaustive: it will try all possible orders of top level method calls.

5.4 Wake Lock Necessity Analysis

With the generated method call sequences, ELITE then identifies checkpoints in each sequence and performs wake lock necessity analysis to detect wake lock misuses.

[4] When starting a service, Android System will also call the onStartCommand() handler after calling onCreate(). We simplified the scenario to ease the presentation.

Identifying checkpoints. Similar to other event-driven programs, an Android app will enter a *quiescent state* after it finishes handling every event [34]. An app may stay quiescent for a long time when there are no new events to handle. Therefore, it is important to analyze such state-transitioning time points, which we refer to as *checkpoints*, to see whether an app unnecessarily holds any wake lock when it enters a quiescent state. As we discussed earlier, at runtime, an app's top level methods are invoked to handle various events. An app will enter a quiescent state after invoking certain top level methods to handle an event.[5] Then, given a sequence of top level method calls $\langle m_1, m_2, \ldots, m_n \rangle$, to identify checkpoints, ELITE first segments it into disjoint non-empty segments $\langle m_1, \ldots, m_i \rangle$ $\langle m_{i+1}, \ldots, m_j \rangle$ $\ldots \langle m_{k+1}, \ldots, m_n \rangle$, each of which contains only the top level method calls that handle one particular event. Since Android's event handling mechanism is well-defined in the API guides [1], the segmentation result is unique. After sequence segmentation, ELITE then performs analysis after the last method call m_{cp} of each segment, and *cp* is the identifier of the checkpoint. For example, for the sequence in Figure 7, ELITE will perform analysis after each method call since each of them handles one event.

Reasoning about the necessity of using wake locks. At each checkpoint *cp* of a method call sequence, ELITE analyzes the dataflow summaries of the methods $m_1, \ldots, m_{cp}$ and performs the following three checks to detect potential wake lock misuses and reports the earliest program points where wake locks should be released:

1) If the app might have acquired a wake lock *wl* before executing the method m_{cp}, is it possible that after executing m_{cp}, *wl* would be released? If yes, ELITE exits with no warnings. Otherwise, ELITE proceeds to the second check.

2) Is it possible that after executing m_{cp}, the app might have stopped all previously started asynchronous (long term) computational tasks? If yes, ELITE reports a warning since the app would not be doing any computation after m_{cp}, but the wake lock *wl* might have been acquired. Otherwise, ELITE proceeds to the third check.

3) Can the asynchronous computational tasks, which might be left running, bring users observable or perceptible benefits? If yes, ELITE exits with no warnings. Otherwise, ELITE reports a warning since the benefit brought by the computation might not justify the energy cost of keeping devices awake. To analyze the benefit of a computational task, ELITE performs API call analysis by leveraging our empirical findings in Section 4.2. Specifically, if any of our observed critical computation APIs is invoked, ELITE decides that the computational task can bring users benefits.

Let us illustrate the analysis process using the example in Figure 7. Suppose that ELITE is performing analysis after the call to `playPause()`. The first check will fail since there is no wake lock releasing operation in `onCreate()` and `playPause()`. Then for the second check, ELITE will find that the music playing task started by `onCreate()` might have been stopped by `playPause()` and therefore the app might unnecessarily hold the wake lock afterwards. It will report a warning accordingly. Finally, after analyzing all checkpoints, ELITE may generate many warnings. In order not to overwhelm users, ELITE will aggregate the warnings by the concerned long term computational tasks and the program points where the wake locks are supposed to be released. To ease issue diagnosis and fixing, when unnecessary wakeup issues are detected, ELITE will also report the locations of the late wake lock releasing operations so that users can consider moving them to more appropriate program points.

5.5 Discussions

ELITE's analysis may not be entirely sound and precise due to common limitations of static analysis. First, the lack of full path sensitivity may cause ELITE to report spurious warnings (false positives or FPs) or miss certain wake lock misuses (false negatives or FNs). Second, although we carefully handled the execution orders of various top level methods, it is hard to guarantee that ELITE can generate and analyze all feasible method call sequences. Third, ELITE analyzes app components separately, assuming each one would take care of its own acquired wake locks. This assumption is reasonable because each app component is a different point through which the Android OS can enter an app [2] and hence it should carefully manage its acquired system resources. However, there is no language-level mechanism to prevent an app component from delegating resource management tasks to other components. If that happens, ELITE may generate FPs and FNs. Lastly, ELITE reasons about whether app computation can bring users perceptible/observable benefits by checking the invocation of prespecified platform APIs. This strategy has shown to be effective [65, 51], but may not work perfectly when the API set is incomplete or app computation involves native code or libraries whose implementation are unavailable for analysis. Although these factors may threaten the effectiveness of ELITE, in our evaluation, we only observed that the lack of full path sensitivity caused ELITE to miss one real issue.

6. EVALUATION

In this section, we evaluate the effectiveness of ELITE. We first introduce some implementation details. We implemented ELITE on top of `soot` [28] and `Apache BCEL` [8]. In `soot`'s `wjpp` phase, ELITE sets all non-abstract methods as analysis entry points to construct whole program call graph. In the `wjtp` phase, ELITE identifies those app components that use wake locks and performs call graph analysis to associate them with their dynamically registered GUI/system event handlers, custom view and asynchronous task callbacks. In this process, ELITE relies on `BCEL` for type resolution (e.g., event listener type) and handling generics (e.g., in `AsyncTask`'s definition [3]). Then in the `jtp` phase, ELITE performs dataflow analysis to summarize top level methods in the app components that use wake locks. Finally, after the `jtp` phase, ELITE generates method call sequences and performs wake lock necessity analysis for issue detection.

6.1 Subjects and Experimental Setup

To study whether ELITE can effectively detect wake lock misuses in real-world Android apps, we conducted experiments using six real issues randomly selected from our identified wake lock misuses. Table 5 lists the basic information of the issues and their containing apps, which cover five different categories (diversity) and have received thousands to millions of downloads (popularity). For our experiments, we selected 12 versions of these apps. For each version, Table 5 lists its revision ID and lines of Java code. Six of the 12 versions contain unnecessary wakeup or wake lock leakage issues and the remaining six are the corresponding bug-fixing versions of these issues. With this setup, we can evaluate the

[5] Some events requires multiple top level method calls to handle, e.g., launching an activity component (see Figure 1) [1].

Table 5: Wake lock misuse detection results of the three techniques under comparison

Index	Subject Information					Issue	ELITE		Relda [38]		Verifier [61]	
	App Name	Category	Downloads	Revision	SLOC	Info.	Warnings	TPs	Warnings	TPs	Warnings	TPs
1	TomaHawk [30]	Music & Audio	5K~10K	b4f339bb24	3,027	Type1	1	1	1	0	2	0
2				883d210525	2,768	Clean	0	0	0	0	0	0
3	Open-GPSTracker [25]	Travel & Local	100K~500K	a9663a7b8d	3,172	Type1	1	1	0	0	1	0
4				6e4e75934b	3,245	Clean	0	0	0	0	1	0
5	MyTracks [24]	Health & Fitness	10M~50M	f2b4b968df	16,186	Type1	2	2	1	0	2	0
6				7749d47238	16,201	Clean	0	0	1	0	0	0
7	FBReader [17]	Books & References	10M~50M	769269336e	53,730	Type2	1	1	0	0	0	0
8				b90c9cdf5f	53,680	Clean	0	0	0	0	0	0
9	MyTracks [24]	Health & Fitness	10M~50M	ca300f0ffd	20,495	Type2	1	1	1	0	0	0
10				17ece1cd75	22,467	Clean	0	0	1	0	0	0
11	CallMeter [11]	Tool	1M~5M	60535cdab9	12,086	Type2	0	0	0	0	1	1
12				4e9106ccf2	12,127	Clean	0	0	0	0	1	0

"Type1" = "unnecessary wakeup"; "Type2" = "wake lock leakage"; "Clean" means that developers fixed the bug in the corresponding version; "TPs" = "true positives".

precision and recall of Elite. In experiments, we configured Elite to generate method call sequences with at most eight top level method calls, which enabled Elite to generate thousands of different sequences for analysis. We also compared Elite with two existing techniques: Relda [38] and Verifier [61]. Relda is a static analysis technique for detecting resource leaks in Android apps [38] (wake locks are also system resources). Verifier is a static analysis technique for verifying that wake locks are properly released at program exit points [61]. We obtained the original implementations of Relda and Verifier from their authors.

6.2 Experimental Results and Analysis

All three techniques finished analyzing each subject in a few minutes on our Linux server and reported some warnings. We checked the warnings against our ground truth and summarize the results in Table 5. Elite detected five real issues, but missed the wake lock leakage in CallMeter. All its reported six warnings are true ones. Relda did not detect real issues. Its reported five warnings are false alarms (Relda is a general resource leak detector, but we only consider its reported wake lock issues here). Verifier reported eight warnings, only one of which is true. To understand why these techniques report FPs and FNs, we further investigated their analysis results and figured out several reasons.

1) Relda and Verifier are **critical computation oblivious**. Their algorithms do not consider the semantics of an app but simply assume that wake locks, once acquired, should be released at a pre-defined set of event handlers. As our empirical findings suggest, Android apps may release wake locks at various program points and such an assumption may not hold in many cases. For instance, in TomaHawk, Verifier expects the wake lock to be released at the `onStartCommand()` handler of `PlaybackService`, regardless whether the wake lock is needed afterwards. This leads to false warnings. As a comparison, Elite makes no assumption on wake lock acquiring/releasing points, but automatically reasons about the necessity of using wake locks at different program points. Therefore, it can precisely infer that `playPause()` is an appropriate wake lock releasing point in `PlaybackService` because the method may stop playing music, which is the critical computation protected by the wake lock. Several other FPs and FNs reported by Relda and Verifier are also due to this reason (░ labelled ones).

2) Some FPs and FNs (▒ labelled ones) are caused by **incomplete handling of program callbacks**. Specifically, Relda and Verifier do not systematically locate all defined program callbacks for each app component and properly handle the temporal relations among them. For example, Relda does not analyze callbacks of asynchronous tasks [3], which are widely-used in real-world Android apps. Therefore, for the wake lock leakage in MyTracks (Figure 6), it can only infer that the `onPause()` handler of the `ExportAllActivity`, which starts the `ExportAllAsyncTask` in its `onCreate()` handler, should take care of wake lock releasing. This is the reason why Relda reports false warnings for both the buggy version and clean version (Rows 9–10). On the other hand, although Verifier can handle asynchronous tasks, it cannot properly handle the temporal relations among their callbacks. For the wake lock leakage in MyTracks, it did not consider that `onPostExecute()` handler may not always follow `doInBackground()`, and therefore failed to detect the issue. As a comparison, Elite does not have such limitations. It adopts a fixed-point iterative algorithm to find all registered GUI/system event handlers, asynchronous task and custom view callbacks for each app component and leverages temporal constraints formulated from Android API guides to generate valid method call sequences for analysis.

3) **The lack of full path sensitivity** in program analysis also led to some FPs and FNs (■ labelled ones). For example, Elite merges dataflow facts at control flow confluence and method return points and therefore failed to detect the wake lock leakage in CallMeter, which occurred because developers forgot to release the wake lock on a specific program path. Relda's analysis is flow-insensitive and also failed to detect the issue. Verifier's analysis considers a certain level of path sensitivity. It does not merge analysis results at method return points and successfully detected the issue in CallMeter. However, Verifier still reports a false warning when analyzing the corresponding bug-fixing version because it cannot handle cases where the lock acquiring and releasing operations are guarded by the same condition.

From the above discussions, we can observe that effectively detecting wake lock misuses in real-world Android apps requires sophisticated algorithms. Elite outperforms existing techniques in precision and recall because it systematically handles program callbacks and considers program semantics by reasoning about the necessity of using wake locks at different program points. Still, Elite's analysis is not path-sensitive and may fail to detect some real issues. We will consider improving its algorithm in future work.

7. THREATS TO VALIDITY

The first threat to the validity of our study results is the representativeness of our selected Android apps. To minimize the threat, we randomly downloaded the latest version of 44,736 apps from Google Play store and chose 31 popular open-source apps from major software hosting sites. We believe our findings can generalize to many real-world Android apps. The second threat is the imprecision of the 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 Soot can generate infeasible call graph edges when resolving virtual calls, because its default CHA algorithm conservatively considers all possible callees based on class hierarchy. To avoid such imprecision, during call graph traversal, if our tool finds that a visited method has multiple callees with the same method signature, it will stop visiting them. We also tried the more precise Spark algorithm of Soot.[6] However, when resolving virtual calls, Spark needs to know the type of the base objects, which requires propagating type information along assignments starting at an allocation site [43]. In Android apps, many objects are not created by invoking constructors, but from factory methods of Android SDK (e.g., the PowerManager and WakeLock objects in Figure 2), whose implementation involves native code. In such cases, Spark will fail to resolve virtual calls (e.g., the wake lock acquiring API call in Figure 2). Although this problem may be solved by manually writing library models or analyzing Android framework code together with each app, such solutions are prohibitively costly. The third threat is the potential errors in our manual investigation of bug reports and code revisions in our empirical study. To reduce the threat, we cross-validated the results and also release them for public access [16]. Lastly, we did not evaluate Elite on many real-world apps. Existing studies [38, 61] conducted evaluations on commercial apps from Google Play store. We did not adopt a similar setting because it is hard to establish the ground truth due to the lack of source code and bug reports for these apps, which can be heavily obfuscated. In our work, we randomly selected five open-source apps and carefully prepared the ground truth for 12 versions of them for experiments. We believe this setting is fair and necessary to compare Elite with existing techniques to analyze their strengths and limitations.

8. RELATED WORK

Our paper relates to a large body of existing studies. This section discusses some representative ones in recent years.

Wake lock misuse detection. Wake lock misuses can cause serious problems in Android apps [46, 48]. Pathak et al. conducted the first study and adapted reaching definition dataflow analysis technique to detect energy bugs caused by wake lock leakage [58]. Later, researchers proposed other static and dynamic analysis techniques for the same purpose [61, 51, 62]. For example, Vekris et al. proposed a static analysis technique for verifying the absence of wake lock leakage in an Android app [61]. Wang et al. designed a technique to repair these issues at runtime [62]. Nonetheless, these studies focused on energy waste issues caused by wake lock leakage. As our study suggests, there are many other common patterns of wake lock misuses that can cause various functional and non-functional issues. These patterns have not been well studied in existing literature.

Resource management and leak detection. Managing wake locks in Android apps resembles managing system resources (e.g., memory blocks, file handles) in conventional software [35, 32, 60]. In recent years, researchers have proposed techniques to facilitate resource management on Android platform. Jindal et al. identified four types of sleep conflicts in Android device drivers and proposed a runtime debugging system to avoid such conflicts [40]. ARSM [41] statically analyzes Android apps to mine resource management specifications (i.e., the correct order of resource operations). Relda adapts the idea of resource safety policy checking [32, 60] to detect resource leaks in Android apps, which cover wake lock leakage [38]. Similar to the earlier verification technique [61], Relda makes assumptions on resource acquiring and releasing points (this is understandable since Relda needs to handle general resources rather than focusing on wake locks) and cannot properly handle the temporal relations among callbacks. Due to such reasons, Relda may fail to detect wake lock misuses in real-world Android apps.

Energy efficiency analysis. Detecting wake lock misuses can help improve energy efficiency of Android apps. There are also studies towards this goal from other angles [50]. For example, vLens [44], eLens [39], eProf [57], PowerTutor [66] can estimate the energy consumption of Android apps to identify hotspots. ADEL [65] and GreenDroid [49] can locate energy bugs caused by ineffective use of program data. Banerjee et al. presented a framework to generate test inputs to uncover energy bugs and hotspots [33]. Li proposed a technique to bundle small HTTP requests in Android apps to save energy [45]. AlarmScope [56] can help reduce non-critical alarm-induced wakeup in Android apps. SEEDS can help developers select energy-efficient implementations of Java libraries for Android apps [53]. GEMMA can generate energy-efficient color palettes for apps running on mobile devices with OLED screens [47]. These techniques are mostly designed for developers. There also exist end user-oriented techniques. For example, eDoctor [52] can correlate system/user events to energy-heavy execution phases to help users troubleshoot battery drains and suggest repairs. Carat [55] shares the same goal, but adopts a big data approach by collecting runtime data from a large community of smartphones to infer energy usage models for providing users advices on improving smartphone battery life.

9. CONCLUSION

In this paper, we conducted a large-scale empirical study to understand how Android developers use wake locks in practice. Our study revealed the common practices of developers and identified eight common patterns of wake lock misuses. To demonstrate the usefulness of our empirical findings, we proposed a static analysis technique, Elite, to detect the two most common patterns of wake lock misuses that can cause serious energy waste. We evaluated Elite on 12 versions of five real-world subjects and the experimental results show that Elite is effective and can outperform two existing techniques. In future, we plan to further improve Elite (e.g., extending it to other six patterns) and conduct more experiments to evaluate its effectiveness.

10. ACKNOWLEDGMENTS

This research was supported by RGC/GRF Grant 611813 of Hong Kong, National Basic Research 973 Program (Grant No. 2015CB352202) and National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491) of China, and 2016 Microsoft Research Asia Collaborative Research Program. We would like to greatly thank the FSE 2016 reviewers for their comments and suggestions that helped improve this work. We would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization of China.

[6] Other algorithms may construct more precise call graphs, but are not efficient for analyzing a large number of apps [63].

11. REFERENCES

[1] Android API guides. https://developer.android.com/guide/.

[2] Android app fundamentals. https://developer.android.com/guide/components/fundamentals.html.

[3] Android AsyncTask. http://developer.android.com/reference/android/os/AsyncTask.html.

[4] Android custom views. http://developer.android.com/training/custom-views/.

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

[6] Android platform version distribution. https://developer.android.com/about/dashboards/.

[7] Android system permissions. http://developer.android.com/guide/topics/security/permissions.html.

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

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

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

[11] CallMeter source code repository. https://github.com/felixb/callmeter/.

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

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

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

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

[16] Elite project website. http://sccpu2.cse.ust.hk/elite/.

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

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

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

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

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

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

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

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

[25] Open-GPSTracker source code repository. https://code.google.com/archive/p/open-gpstracker/.

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

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

[28] Soot project website. http://sable.github.io/soot/.

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

[30] TomaHawk source code repository. https://github.com/tomahawk-player/tomahawk.

[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[32] M. Arnold, M. Vechev, and E. Yahav. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *OOPSLA*, pages 143–162, 2008.

[33] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting Energy Bugs and Hotspots in Mobile Apps. In *FSE*, pages 588–598, 2014.

[34] E. Cheong and J. Liu. galsC: A Language for Event-Driven Embedded Systems. In *DATE*, pages 1050–1055, 2005.

[35] J. Clause and A. Orso. LEAKPOINT: Pinpointing the Causes of Memory Leaks. In *ICSE*, pages 515–524, 2010.

[36] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *CCS*, pages 627–638, 2011.

[37] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-oriented Languages. In *OOPSLA*, pages 108–124, 1997.

[38] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and Detecting Resource Leaks in Android Applications. In *ASE*, pages 389–398, 2013.

[39] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *ICSE*, pages 92–101, 2013.

[40] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: Understanding and Treating Sleep Conflicts in Smartphones. In *EuroSys*, pages 253–266, 2013.

[41] Y. Kang, X. Miao, H. Liu, Q. Ma, K. Liu, and Y. Liu. Learning Resource Management Specifications in Smartphones. In *ICPADS*, pages 100–107, 2015.

[42] M. Lan, C. Tan, J. Su, and Y. Lu. Supervised and Traditional Term Weighting Methods for Automatic Text Categorization. *IEEE TPAMI*, 31(4):721–735, 2009.

[43] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using SPARK. In *CC*, pages 153–169, Berlin, Heidelberg, 2003.

[44] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating Source Line Level Energy Information for Android Applications. In *ISSTA*, pages 78–89, 2013.

[45] D. Li, Y. Lyu, J. Gui, and W. G. Halfond. Automated Energy Optimization of HTTP Requests for Mobile Applications. In *ICSE*, pages 249–260, May 2016.

[46] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In *MSR*, pages 2–11, 2014.

[47] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach. In *FSE*, pages 143–154, 2015.

[48] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How Developers Detect and Fix Performance Bottlenecks in Android Apps. In *ICSME*, pages 352–361, 2015.

[49] Y. Liu, C. Xu, and S. C. Cheung. Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. In *PerCom*, pages 2–10, 2013.

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

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

[52] 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*, pages 57–70, 2013.

[53] I. Manotas, L. Pollock, and J. Clause. SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In *ICSE*, pages 503–514, 2014.

[54] P. Mutchler, Y. Safaei, A. Doupé, and J. Mitchell. Target Fragmentation in Android Apps. In *MoST*, May 2016.

[55] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *SenSys*, pages 10:1–10:14, 2013.

[56] S. Park, D. Kim, and H. Cha. Reducing Energy Consumption of Alarm-induced Wake-ups on Android Smartphones. In *HotMobile*, pages 33–38, 2015.

[57] 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 *EuroSys*, pages 29–42, 2012.

[58] 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 *MobiSys*, pages 267–280, 2012.

[59] X. Perez-Costa and D. Camps-Mur. IEEE 802.11E QoS and Power Saving Features Overview and Analysis of Combined Performance. *IEEE WC*, 17(4):88–96, 2010.

[60] E. Torlak and S. Chandra. Effective Interprocedural Resource Leak Detection. In *ICSE*, pages 535–544, 2010.

[61] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal. Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs. In *HotPower*, pages 3–3, 2012.

[62] X. Wang, X. Li, and W. Wen. WLCleaner: Reducing Energy Waste Caused by WakeLock Bugs at Runtime. In *DASC*, pages 429–434, 2014.

[63] X. Xiao and C. Zhang. Geometric Encoding: Forging the High Performance Context Sensitive Points-to Analysis for Java. In *ISSTA*, pages 188–198, 2011.

[64] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *ICSE*, pages 89–99, 2015.

[65] 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 *CODES+ISSS*, pages 363–372, 2012.

[66] 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 *CODES+ISSS*, pages 105–114, 2010.

[67] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *CCS*, pages 1105–1116, 2014.