What is keeping my phone awake?  
Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps

Abhinav Pathak  
Purdue University  
pathaka@purdue.edu

Abhilash Jindal  
Purdue University  
jindal0@purdue.edu

Y. Charlie Hu  
Purdue University  
ychu@purdue.edu

Samuel P. Midkiff  
Purdue University  
smidkiff@ecn.purdue.edu

ABSTRACT
Despite their immense popularity in recent years, smartphones are and will remain severely limited by their battery life. Preserving this critical resource has driven smartphone OSes to undergo a paradigm shift in power management: by default every component, including the CPU, stays off or in an idle state, unless the app explicitly instructs the OS to keep it on! Such a policy encumbers app developers to explicitly juggle power control APIs exported by the OS to keep the components on, during their active use by the app and off otherwise. The resulting power-encumbered programming unavoidably gives rise to a new class of software energy bugs on smartphones called no-sleep bugs, which arise from mishandling power control APIs by apps or the framework and result in significant and unexpected battery drainage.

This paper makes the first advances towards understanding and automatically detecting software energy bugs on smartphones. It makes the following three contributions: (1) we present the first comprehensive study of real world no-sleep energy bug characteristics; (2) we propose the first automatic solution to detect these bugs, based on the classic reaching definitions dataflow analysis algorithm; (3) we provide experimental data showing that our tool accurately detected all 12 known instances of no-sleep bugs and found 30 new bugs in the 86 apps examined.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Design, Experimentation, Measurement

1. INTRODUCTION
1.1 Motivation
Smartphones have surpassed desktop machines in sales in 2011 to become the most prevalent computing platforms [1]. To enrich the user experience, modern day smartphones come with a host of hardware I/O components embedded in them. The list of components broadly fall into two categories: traditional components such as CPU, WiFi NIC, 3G radio, memory, screen and storage that are also found in desktop and laptop machines, and exotic components such as GPS, camera and various sensors. And they differ from their desktop/laptop counterparts in that power consumed by individual I/O components is often comparable to, or higher than, the power consumed by the CPU.

This, along with the fact that smartphones have limited battery life, dictates that energy has become the most critical resource of smartphones. Preserving this crucial resource has driven smartphone OSes to resort to a paradigm shift in component power management. On desktop machines, where the CPU accounts for a majority of the energy consumption, the default energy management policy is that the CPU stays on (or runs at a high frequency) unless an extended period of low load has been observed. The policy is consistent with the historical notion that energy management is a second class citizen since machines are plugged into a power source [2]. Smart phones, in sharp contrast, make power management policy a first class citizen. In fact, the power management policy on smartphones has gone to the other extreme: the default power management policy is that every component, including the CPU, stays off or in an idle state, unless the app explicitly instructs the OS to keep it on!

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2012 ACM 978-1-4503-1301-8/12/06 ...$10.00.
phone apps, since an app may be performing critical tasks by inter-
mittently interacting with the external world using various sensors.
For example, an app syncing with a remote server over the network
may appear to perform no activity when waiting for the server to
send its reply, and the system may be put to sleep by the aggres-
sive sleeping policy, leaving the remote server with a view of lost
connectivity.

To avoid such disruptions due to the aggressive sleeping policy,
smartphone OSes provide a set of mechanisms for app developers
to explicitly notify the OS of their intention to continue using each
component. In particular, the OS exports explicit power manage-
ment handles and APIs, typically in the form of power wakelocks
and acquire and release APIs, for use by the app developer to
specify when a particular component needs to stay on, or awake,
until it is explicitly released from duty.

We argue such explicit management of smartphone components
by app developers has presented to the app developer a profound
paradigm shift in smartphone programming that we call as power-
encoded programming. This new programming paradigm
places a significant burden on developers to explicitly manipulate
the power control APIs (details one such example of the
burden placed on developer due to power encumbrance). This ma-
nipulation is required to ensure the correct operation of the apps.
Consequently, power-encumbered programming unavoidably gives
rise to a new class of software energy bugs on smartphones, called
no-sleep bugs. No-sleep bugs are defined as energy bugs resulting
from mis-handling power control APIs in an app or framework,
resulting in the smartphone components staying on for an unneces-
sarily long period of time. No-sleep bugs form one important cat-

1.2 Our Contributions

This paper takes the first steps towards understanding and au-
tomatically detecting software energy bugs on smartphones. Specifi-
cally, our paper makes three concrete contributions:

(1) The first characterization study of no-sleep energy bugs in
smartphone apps: We present the first comprehensive real world
no-sleep energy bug characterization study. Our study is based on
no-sleep energy bugs in real world apps and the Android frame-
work, including popular apps (e.g., Facebook) and built-in (i.e.,
shipped with) apps and services (e.g., the Android email app). The
bugs are collected by crawling Internet mobile forums, bug repos-
itories, commit logs of open source Android apps and by running
our no-sleep bug detector developed in this paper. For each bug,
we carefully examine its reported symptoms, corresponding source
code and related patches (when available), and developer’s discus-
sions (when available), or the analysis performed by our bug de-
tector. Our study reveals a taxonomy of three major causes of no-
sleep energy bugs, which provide useful guidelines and hints to
designing effective detection techniques. Our study also confirms
the significant burden power-encumbered programming places on
app developers. For example, making a single outgoing or incom-
ing phone call in Android involves about 40 invocations of power
control APIs in the Dialer app and the framework’s Radio Inter-
face Layer services to dynamically manage the power control of
the CPU, screen, and other sensors!

(2) The first solution to automatically detect no-sleep energy
bugs: We make the key observation that power control APIs are
explicitly embedded in the app source code by the app developers,
and two out of the three causes for no-sleep energy bugs from our
characterization study are because a turn-on API call is missing a
matching turn-off API call before the end of the program execu-
tion. We thus propose a compile-time solution based on the classic
reaching definitions dataflow analysis problem to automatically
infer the possibility of a no-sleep bug in a given app. Our solution
detects no-sleep bugs in single-threaded and multi-threaded apps,
as well as event-based apps which have multiple entry points. Like
all static analysis based tools, our detection tool can suffer false
positives but has the tremendous advantage of no runtime overhead
and no false negatives (to the best of our abilities of establishing
the ground truth).

We further present the complete implementation of our static
analysis detection tool for apps written for Android. The tool is
capable of running directly on the app installers (.apk files) and
hence source code is not required. Our implementation handles
the specifics of event-driven mobile programming and of the Java
language such as runtime null pointer exceptions and object refer-
ences.

(3) Detecting new no-sleep bugs in Android apps and framework:
We have run our no-sleep bug detection tool on 86 Android apps
and the framework collected from the Android market. Experiment-
tal evaluation shows that our tool accurately detected all reported
instances of no-sleep bugs, as well as 30 instances of new previ-
ously unreported no-sleep bugs. These include no-sleep bugs in
many popular apps, e.g., the default Android Email app. Our no-
sleep bug detection incurred false positives in 13 out of the 55 apps
it reported to contain a bug.

2. POWER ENCUMBERED
PROGRAMMING

In this section, we first describe the energy management APIs
and their semantics that are exposed to the developers in the An-
droid smartphone OS, and discuss the burden they impose on the
app developers. We first discuss programming APIs for traditional
components (e.g., screen, CPU) and then for exotic components
(e.g., GPS, Camera). We also discuss the issues arising from event
based programming model of smartphone apps. We then introduce
the prominent class of energy bugs studied in this paper: no-sleep
bugs.
### 2.1 Managing Traditional Components

The Android framework exports wakelock functionality through the `PowerManager.WakeLock` class, with 4 different options and associated APIs for managing several traditional components: CPU, screen, and the keyboard backlight. A wakelock is an instance (object in Java) of the wakelock class, instantiated using one of 4 options, and each option has a different effect on the hardware component, as summarized in Table 1. For example, option `FULL_WAKE_LOCK` instantiates a lock that when acquired both keeps the CPU and screen on at full brightness and turns on the keyboard backlight.

Listing 1 illustrates a basic wakelock usage: how to ensure that the CPU does not sleep during some critical phase. The app declares a wakelock (`pm` is an instance of PowerManager) and then acquires it, which instructs the OS not to put the CPU to sleep, irrespective of user activity, since it intends to perform some critical task. Once the critical task (in this case a remote network sync) is completed the app releases the wakelock, indicating to the OS that CPU can now sleep according to its sleeping policies.

<table>
<thead>
<tr>
<th>Component lock/manager name (API to start/stop)</th>
<th>Component(s)</th>
<th>Battery Drain up to (%/hr)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PARTIAL_WAKE_LOCK(acquire/release)</code></td>
<td>CPU</td>
<td>5%</td>
<td>CPU runs despite any timers</td>
</tr>
<tr>
<td><code>SCREEN_DIM_WAKE_LOCK(acquire/release)</code></td>
<td>CPU and Screen (DIM)</td>
<td>15%</td>
<td>No illumination if shutdown, else illuminates till lock release (Flag <code>ACQUIRE_CAUSES_WAKEUP</code>) forces illumination in all cases)</td>
</tr>
<tr>
<td><code>SCREEN_BRIGHT_WAKE_LOCK(acquire/release)</code></td>
<td>CPU and Screen (bright)</td>
<td>25%</td>
<td></td>
</tr>
<tr>
<td><code>FULL_WAKE_LOCK(acquire/release)</code></td>
<td>CPU, Screen (bright) and Keyboard backlight</td>
<td>25%</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1: Summary of power operations exported by Android APIs.**

Exotic Components

- **PROXIMITY_SCREEN_OFF_WAKE_LOCK(acquire/release)**: Screen, Proximity Sensor 25% Screen shuts if sensor activates
- **LocationManager(requestLocationUpdate/removeUpdates)**: LocationManager 15% Tracks user location
- **SensorManager(registerListener/unregisterListener)**: Accelerometer, Gyro, Proximity, Magnetic Field, etc. U.S. 10% Sensormanager class controls various sensors on phone
- **MediaRecorder(start/stop)**: Mic/Camera (for video) 20% Usually stores media on sdcard
- **Camera(startPreview/stopPreview)**: Camera (for still pictures) 20% One app at a time registers camera

Listing 1: An example power wakelock usage.

```java
PowerManager.WakeLock wl = pm.newWakeLock(PowerManager.
PARTIAL_WAKE_LOCK);
wl.acquire(); //CPU should not go to sleep
net_sync(); //Perform critical task here
wl.release(); //CPU is free to sleep
```

#### 2.2 Managing Exotic Components

In addition to traditional components, modern smartphones come with several "exotic" components embedded in them. These include GPS, camera, several sensors such as an accelerometer, proximity sensor, and gyroscope. Some of these components are the

- **GPS**
- **Camera (for still pictures)**
- **Screen, Proximity Sensor**
- **CPU, Screen (DIM) and Keyboard backlight**
- **CPU and Screen (bright)**

In contrast, reference counted wakelocks are like semaphores. Each acquire() of a wakelock increments the internal counter associated with the (instance of) wakelock, and a release() decrements the internal counter. It only lets the component sleep (from the perspective of this wakelock) if the internal counter value reaches 0. A release is a nop in other cases.

To make matters even more complicated, an acquire can also be called with a `timer`, which instructs the system to release it automatically once the timeout interval expires.

Third, the above power control semantics are from the perspective of one wakelock. Unlike traditional mutual exclusion locks, different wakelocks (even instantiated with different options) on the same component can be held by multiple entities (e.g., processes and threads) in the system at the same time. Even a single entity may hold multiple (instances of) wakelocks on the same component.

The power control effect on a component must take into account the state of all wakelocks. The component is switched on when the first wakelock is held. Only when all the wakelocks from all the entities for the component are released, taking into account the reference counting semantics for each, can the component go to sleep, subject to higher level sleeping policies (administered by framework processes), e.g., sleep after 5 seconds of user inactivity.

This demonstrates the new programming burden inflicted on app developers: power management is no longer just a transparent OS or driver task; the developers now need to perform explicit power management in the app layer.

#### 2.3 Testing

It’s important to test each component to ensure that it’s working correctly and efficiently. This includes testing the acquisition and release of the wakelock, ensuring that the component is properly activated and deactivated, and that the battery drain is minimized.

Apps need android.permission.WAKE_LOCK permission from users to use this class.
biggest energy consumers in smartphones, e.g., GPS and camera, and drain the battery at a high rate [11, 12, 13]. Unlike some of the traditional components, e.g., WiFi NIC, the new exotic components are used in an explicit on-off fashion. For example, the GPS is explicitly turned on, using the OS exported API, to acquire the smartphone location and in this state it consumes battery at a high rate. Once the location is determined, the component is explicitly turned off, triggering the component to return to a low power state.

Like wakelocks, the explicit power management of exotic components places a significant programming burden on app developers. An incorrect or inefficient use of these APIs can easily lead to poor utilization of these components, wasting significant battery energy (see §2).

Semantics: Table 1 lists the APIs exported by Android for accessing the exotic components. Their semantics of power control of these components is similar to the plain wakelocks in §2, i.e., no reference counting or timer-based release.

In summary, the developers are burdened with explicitly manipulating power control APIs for both traditional and exotic smartphone components to ensure the correct operations of the apps. We call this new smartphone programming paradigm as power-encumbered programming.

2.3 Issues from Event-based Programming

The complexity of power-encumbered programming is exacerbated by the event-based nature of smartphone apps. Compared to programming in desktop/server environments, smartphone programming is event-oriented because of the inherent interactive nature of phone apps. A typical user-facing smartphone app is written as a set of event handlers with events being user or external activities. The developer needs to keep track of each possible event and when it may be triggered, and manipulate the wakelocks accordingly.

We illustrate how the level of complexity introduced by power-encumbered programming is exacerbated by event-based programming through a concrete example from the Dialer app that comes with the Android framework.

The Dialer app implements the dialing functionality of the phone. The app is triggered when the user receives an incoming call or when the user clicks the phone icon to make an outgoing call. To implement its functionality, the app explicitly maintains three wakelocks: FULL_WAKE_LOCK for keeping the screen on (e.g., in situations like when the user is dialing the numbers to call), PARTIAL_WAKE_LOCK for keeping the CPU on (e.g., in case of an incoming call when the phone is switched off), and PROXIMITY_SCREEN_OFF_WAKE_LOCK which switches the proximity sensor on and off (to detect user’s proximity to the phone).

To manage the three wakelocks, the app explicitly maintains a state machine where the states represent the lock behavior, i.e., which lock needs to be acquired and which needs to be released, and the “condition” of the phone represents the state transitions. The conditions are diverse and include events such as (a) if the phone gets a call, (b) if the phone is pressed against the user’s ear in which case the proximity sensor triggers the screen to go off, (c) if the call ends, (d) if a wired or Bluetooth headset is plugged in (e.g., in the middle of a call), (e) if the phone speaker is turned on, (f) if the phone slider is opened in between calls, and (g) if the user clicked home button in the middle of a call. For each of these triggering events, the phone changes the state of wakelock state machine, acquiring one and releasing another.

In addition to wakelocks in the Dialer app, the Radio Interface Layer (RIL) in Android maintains additional 5 wakelocks to handle incoming and outgoing calls. Using explicit component access tracing through an instrumented Android framework running on a Google Nexus One handset, we observed that performing a single outgoing or incoming call in Android resulted in 30-40 distinct instances of wakelock acquirements and releases!

2.4 No-Sleep Bugs

The new burden of explicit component power manipulation from power-encumbered programming, combined with the complexity of handling events in the app behavior, can easily overwhelm developers and lead to programming mistakes in manipulating the power control APIs. Incorrect or inefficient usage of such APIs can lead to an unexpected drain of the phone battery, known as no-sleep bugs [14].

A “no-sleep bug” is a condition where at least one component of the phone is woken up and is not put to sleep due to a mistake in manipulating power control APIs in an app. The component that is woken up continues to drain the battery for a prolonged period of time, resulting in severe and unexpected battery drain. Typically the battery drain continues until the app is forcefully killed or the system is rebooted.

No-sleep bugs form one of the most important categories of software energy bugs in smartphone apps [14]. Unlike regular software bugs in apps, energy bugs do not lead to an app crash or OS blue screen of death. An app hit by an energy bug continues to provide the intended functionality, with a single difference: the phone suffers a severe, unexpected battery drain. The severity of the energy drain due to the bug depends on the component that is not put to sleep. As shown in Table 1 for each of the 3 components (GPS, Screen with full or low brightness and camera), the impact of a no-sleep bug can be severe with the battery draining at a rate of 10-25% every hour without any user interaction.

For other components, e.g., the CPU and proximity sensor, the battery drains at a relatively low rate - up to 5% every hour. When a CPU wakelock (PARTIAL_WAKE_LOCK) is held, it prevents the CPU from ‘freezing’, a state where it would consume zero power (IDLE state). In a wakelock held state, the CPU draws minimal power (depending on the CPU specifications of the handset). However, as the CPU remains on, other activities continue to run, e.g., WiFi NIC chatters, background periodic OS processes, hardware interrupts handling by OS, etc. These activities together consume, as measured on Google Nexus One, about 5% of the battery every hour. Any additional user activity is not accounted for in this. As a result, over a long period of time, say 12 hours, an only-CPU wakelock bug can drain about 50-60% of the battery without any user interaction or performing necessary activities.

3. METHODOLOGY

To characterize the root cause of no-sleep bugs observed in current mobile apps, we collected no-sleep bugs in smartphone apps in four ways. (a) Mobile forums: We crawled 4 popular mobile Internet forums (the same as in [14]), one general forum with discussions covering all mobile devices and OSes, and three OS/company specific mobile forums. In total we crawled 1.2M posts, from which we filtered out posts related to no-sleep bugs in smartphone apps. For each app reported by the user to contain a no-sleep bug, we downloaded the binary installers of the versions of the app that was reported to contain the bug and the first version which had the bug.

In Android, apps (esp. background services) are not killed; they run in background once a user stops interacting with them. They are usually killed by the system only in case of memory pressure.

5These rates were calculated using HTC nexus and magic handsets running Android. A fully charged battery holds between 1100mAH to 1500mAH of charge.
problem not put to sleep. Listing 2 shows a code-snippet that represents a typical template of a no-sleep bug where an app takes a different, somewhat unanticipated code path after waking up a component and therefore does not put it back to sleep. As in Listing 1, the critical task in the app, net_sync(), is protected by acquiring and releasing the CPU wakelock instructing the CPU not to sleep during the remote syncing phase. However, routine net_sync() may throw exceptions, a Java language mechanism for notifying apps of some failure conditions, such as a connect to a remote end host failed, a string could not be parsed to integer, or a specific file to be opened does not exist. A thrown exception is explicitly caught by the try catch block which simply prints the exception for debugging purposes. Now the no-sleep bug can manifest itself in the following code path. First the try block executes and acquires the wakelock. Next a call is issued to the critical task, net_sync(). If an exception is raised inside net_sync(), the control directly jumps to the catch block, the debug output is printed, and the code exits the try catch block. Consequently, the code-path followed does not release the wakelock, keeping the CPU on indefinitely. To fix this problem, the wakelock should be released in the finally block so that it is always executed.

A large number of no-sleep bugs are caused by this second reason. These include popular apps such as FaceBook [22], Agenda widget [21] (another version), MyTrack [25] (no sleep of GPS), BabbleSink [20], CommonsWare and apps [27], as well as services that come with the Android framework, such as Android Telephony [28], Android Exchange [29, 30], and WifiService [31]. For example, in SIP service [19], the wakelock was not released since the objects containing the wakelocks were deleted and so the lock handlers were deleted along with them.

The third cause for a no-sleep code path is that a higher level condition (like an app level deadlock) prevented the execution from reaching the point where the wakelock was to be released. This is likely to happen in smartphone programming because event-based programming of smartphones can lead to many possible code branches (as in the example of the Android Email app) that makes it difficult for the programmer to anticipate all the possible code paths and keep track of the wakelock state. LocationListener [33] in the Android framework contained such a no-sleep bug. The developer did release the wakelock, however, a higher level app deadlock prevented the code from entering the release phase of the app.

Finally, the most common pattern of no-sleep code-path bug is a result of the fact that developers do not properly understand the life-cycle of Android processes. In Android, an app activity once started is always alive. When the user exits any app, Android saves the state of the app and passes it back to the app if the user returns to it. The app is only completely killed when the phone is critically low on RAM or when the app kills itself. This methodology is used to reduce the startup time of the app and to maintain its state.

This essentially means that the app may not actually be destroyed for very long periods of time. But many app developers only re-
Table 2: No-sleep bug case study. Entries with (F) represent bugs in the Android framework, and with (*) represent new bugs found by our technique.

<table>
<thead>
<tr>
<th>App</th>
<th>Description</th>
<th>Bug Description</th>
<th>Drain</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agenda Widget</td>
<td>Popular Android widget managing news/calendar</td>
<td>Two bugs were reported in different versions (a) not all branches release wakelocks in AlarmService; (b) programmer forgot to call wakelock release after acquiring it.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>FaceBook</td>
<td>The default FaceBook App v 1.3.0</td>
<td>facebook.katana.HomeActivity, the central Activity, acquires wakelock to run FaceBookService. Not all possible branches in the service release wakelocks.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>k9mail</td>
<td>One of the most popular email client for Android</td>
<td>Per-thread wakelock maintained. Wakelock acquired when IMAP DONE was sent, but was not released in MessagingControllerPushReceiver during IDLE stat.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>CheckinMaps</td>
<td>Visual stories on maps</td>
<td>GPS remains on, even when user closes the app (calling the onPause () handler).</td>
<td>5 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>(*) MyTrack</td>
<td>Track User Path online</td>
<td>GPS remains on, even after user navigates away from the app draining battery.</td>
<td>5 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>BubbleSink</td>
<td>Find phone’s location</td>
<td>A NullPointerException causes the thread to exit without releasing wakelock.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>CommonsWare</td>
<td>Android Training Book</td>
<td>Wakelock released without finalize.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>Sip Service (F)</td>
<td>Std. voice protocol implementation in Android</td>
<td>A Sip handler (object) was deleted which had wakelocks acquired before releasing the wakelock. The deleted handlers cannot perform release() to release the wakelocks.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>Telephony (F)</td>
<td>Telephony Handler: RIL service in Android code</td>
<td>Android telephony does not release the partial wakelock right away if there is an error in sending the RIL request, preventing the phone to go in power collapse draining battery.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>(*) Android Exchange (F)</td>
<td>The default email app in Android framework</td>
<td>During background syncing of mailboxes in an exchange account, the app acquires wakelock and does not release in all failure conditions, specifically in IOException.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>WifiService (F)</td>
<td>Android WiFi Handlers</td>
<td>CPU does not go to sleep during a message removal and wakelock was held forever.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>PowerManager(F)</td>
<td>PowerService Android</td>
<td>Two instances of wakelocks are not released in PowerManagerService in Android.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>LocationListener (F)</td>
<td>GPS handling library in Android framework</td>
<td>A deadlock in LocationManagerService for releasing wakelocks after client notifications have been received prevented the release of the wakelocks draining battery.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>(*) Android Email app (F)</td>
<td>Default Android email app performing sync</td>
<td>Race condition between email synchronizing thread and the main thread which kills the synchronizing thread resulted in a shared wakelock to remain in acquired state after exit.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>MyTrack</td>
<td>Track User Path online</td>
<td>Wakelock acquired and released much before and after the required functionality in app.</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>GoogleBackup (F)</td>
<td>Cloud backup</td>
<td>Wakelock reported to be held for a long duration of time (up to an hour) in poor network.</td>
<td>11 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>GPS Driver (F)</td>
<td>Android GPS handler</td>
<td>Wakelocks are being held for longer than needed in low level GPS driver code.</td>
<td>15 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>Google Maps</td>
<td>Android Google maps</td>
<td>App was reported to hold wakelock for several hours even when it was not used.</td>
<td>10 hrs</td>
<td>📌</td>
</tr>
</tbody>
</table>

No-Sleep Dilution

<table>
<thead>
<tr>
<th>App</th>
<th>Description</th>
<th>Drain</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyTrack</td>
<td>Track User Path online</td>
<td>9 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>GoogleBackup (F)</td>
<td>Cloud backup</td>
<td>11 hrs</td>
<td>📌</td>
</tr>
<tr>
<td>GoogleMaps</td>
<td>Android Google maps</td>
<td>10 hrs</td>
<td>📌</td>
</tr>
</tbody>
</table>

Listing 3: Wakelock Complexity

```java
@override
protected void finalize()
{
    /**
     * It is understood that This finalizer is not
     * guaranteed to be called and the release lock
     * call is here just in case there is some path
     * that doesn’t call onDisconnect and or
     * onConnectedInOrOut.
     */
    if (mPartialWakeLock.isHeld()) {
        Log.e(LOG_TAG, "[CdmaConn] UNEXPECTED: mPartialWakeLock
         is held when finalizing.");
        releaseWakeLock();
    }
}
```

lease the wakelock in the onDestroy() call-back, instead of in onPause(). onDestroy() is only called when the app component is about to be destroyed. As a result, once an app with this bug is started, the phone will only sleep when it is running critically low on memory (which may take a long period of time).

Manually tracking all possible code paths for wakelock acquire/release appears to be a daunting task for app developers. Listing 3 shows an example of the complexity involved in power-encumbered programming. This is a code-snippet from the Android framework’s class CdmaConnection. This class uses a PARTIAL_WAKE_LOCK for managing the connection and releases the wakelock when the connection is disconnected. However, there are many different possible program paths arising from different patterns of user interactions, hardware states dependent on the external environment, etc. The developer included releaseWakeLock in finalize as an additional safety measure, even though finalize is not guaranteed to be called. This example shows the need for an automated tool that can aid developers in checking all possible program paths for no-sleep bugs.

4.2 No-Sleep Race Condition

The second category of no-sleep bugs we observed was cause by race conditions in multi-threaded apps. Specifically, we observed that the power management of a particular component was carried out (i.e., switched on and off) by different threads in the app. In the common case, one thread switches the component on, and some time later another thread switches the component off, resulting in the normal behavior of component utilization. However, in a corner case condition, it can happen that the thread that switches on the component gets scheduled to run after the thread that switches the component off, resulting in a no-sleep bug with the component left...
Listing 4: No-sleep bug: race condition.

```java
public void Main_Thread()
{
    mKill = false; //Unset kill flag
    wl.acquire(); //CPU should not sleep
    start(worker_thread); //Start worker
    //...Do Something
    mKill = true; //Set kill flag
    stop(worker_thread); //Signal worker
    wl.release(); //CPU can sleep now
    sleep(180000); //Sleep for 3 minutes
    stop(worker_thread); //CPU should not sleep
} //End Main_Thread()

public void Worker_Thread()
{
    while(true)
    {
        if(mKill) break; //Break if flagged
        net_sync(); //Critical task
        wl.release(); //Rel. wl before sleep
        sleep(180000); //Sleep for 3 minutes
        wl.acquire(); //CPU should not sleep
    } //End while loop;
} //End Worker_Thread;
```

... on. Effectively there is a race condition between the manipulation of the wakelock by the two threads.

Listing 4 shows a code snippet of a no-sleep bug caused by a race condition. Main_Thread runs first, acquiring a wakelock (waking up a component, e.g., the CPU), and then fires Worker_Thread which periodically executes a critical task, e.g., syncing stock updates. After every sync Worker_Thread gives up the lock, sleeps for 3 minutes (allowing the CPU to sleep), and re-acquires the lock after waking up. This process is repeated in an infinite loop until Worker_Thread is notified by Main_Thread using the mKill flag to break out of the loop. To initiate the termination of the app, Main_Thread sets the mKill flag, and calls the API stop to signal Worker_Thread to initiate the halt which wakes Main_Thread up if it is in sleep state. Main_Thread releases the wakelock after calling stop.

In the normal scheme of things, the code in Listing 4 executes without any energy bug. However, consider the following sequence of events. Main_Thread sets the mKill flag, signals Worker_Thread to stop, and releases the wakelock. Then Worker_Thread wakes up, acquires the lock and exits the loop because of the mKill flag. As a result, the wakelock remains held by the app and is never released. A key point to note here is that the semantics of the stop() API called by Main_Thread does not guarantee that the return from the call will be synchronized, i.e., only after Worker_Thread exits. Had that been the case, there would have been no race condition and hence no no-sleep bug in the app.

Tracing no-sleep bugs in app source code caused by race conditions is particularly hard since it requires enumerating all the possible execution orderings of the threads. However, using our automatic techniques for detecting no-sleep bugs presented in [3], we were able to detect an instance of a no-sleep bug caused by a race condition in the Android Email App [35, 36, 42], which had a similar pattern as shown in Listing 4.

### 4.3 No-Sleep Dilation

This category of no-sleep bugs differs from the first two categories in a single aspect: the component woken up by the app is ultimately put to sleep by the app, but only after a substantially longer period of time than expected or necessary. For example, consider the code in Listing 5. Suppose routine net_sync() usually finishes in a few seconds, but during a particular run it hangs for an unexpected length of time before it returns, the battery is drained for that prolonged period of time.

While it is arguable that keeping the system on during the execution of a critical task, no matter how long it takes, was indeed the intention by the app developer, we characterize such situations as the third category of no-sleep energy bugs, no-sleep dilation. These are considered no-sleep energy bugs for the following reasons: (a) such instances of prolonged component wakeup are usually unexpected, even by the app developer, as we found by reading the log of code commits; (b) the mobile programming API documentation strictly warn developers not to keep the components awake for prolonged periods of time unless it is actually required, e.g., in the Skype app, where a user performing a video call requires the components (screen, CPU) to be switched on from the start till the end of the call irrespective of how long the call persists; (c) instances of no-sleep bugs in this category were observed to cause severe frustration among smartphone users since the energy drain was both severe and unexpected; and (d) the root cause of such prolonged completion time of critical tasks was usually because of a higher level bug (i.e., programming mistake) in the code, which significantly inflated the running time of critical task.

We found two causes for no-sleep dilation in smartphone apps: app delay and app optimizations. We first discuss the dilation caused by app delay in the GPS driver [39] in Android. The driver held wakelocks for longer than needed. In some circumstances, after holding the wakelock, the driver issued a wait, waiting for an event. However, after being signaled, a second wait was issued causing another wait until the driver was signaled again. All this was done while holding a wakelock. As a result, a higher level bug in handling signals extended the time period the wakelock was being held.

Another cause of no-sleep dilation observed in the apps we studied results from poor placement of component wakeup code in the app. For example, consider the code in Listing 6. The dilation could happen if the app developer, instead of just protecting the critical part `net_sync()`, wrapped a large piece of code in wakelocks. We observed such a bug in the MyTrack app [25] where the developer acquired the CPU wakelock the moment the app was turned on and released it when the app completed. However, the critical part of the code was only the period where the user clicked the track button for location tracking.

### 5. DEBUGGING NO-SLEEP BUGS

Two general approaches exist to understanding program behavior: those done at compile-time and those done at run-time. Compile-time approaches incur no run-time overhead. While a run-time approach can gather perfect, or near perfect information about a given run, a compile-time approach will (conservatively) determine facts that may be true on any run. Because of the runtime overhead, compile-time approaches are preferred when they are sufficiently accurate, as is the case with our problem. In this paper, we present a static, compile-time solution for detecting no-sleep energy bugs in smartphone apps.

Our solution treats the acquire and release of a wakelock $l$ as a definition of (assignment to) the variable $v_l$ corresponding to $l$. A definition $d$ of a variable $v_l$ is said to reach some point $p$ in a program if there exists a path from $d$ to $p$ that does not redefine $v_l$. Therefore, if a definition of $v_l$ corresponding to acquiring a wakelock reaches the end of some code region there exists a no-sleep code path in the region. Thus, detecting no-sleep code paths corresponds exactly to the reaching definitions (RD) dataflow problem [5], which can be solved by a standard compile-time dataflow analysis.

---

8 We assume that the `sleep` API call used in Listing 5 registers a wake-up timer with the CPU, which in case the phone freezes during the app sleep, wakes up the CPU at the end of the timeout.
We first present, in [6], our solution when only a single thread is being analyzed, and then, in [7] show how to apply the problem to multi-threaded smartphone apps to detect no-sleep bugs arising from races. We leave detecting and debugging no-sleep dilation bugs as future work.

6. NO-SLEEP CODE PATHS

We first give an overview of dataflow analysis, and then describe our solution as a dataflow analysis problem.

6.1 Dataflow Analysis: An Overview

Dataflow analysis refers to a set of techniques that ascertain facts about program properties by analyzing the effects of statements along different paths of a control flow graph (CFG) on those properties. There exist many useful dataflow analysis, e.g., RD (discussed above), live variable analysis (which variable values are used after a block), and available expressions (which subexpressions have already been computed and are unchanged yet).

Each node in a CFG is a basic block of statements, i.e., the block has exactly one entry point and one exit point. There exist a directed edge (B1, B2) in the CFG connecting every pair of blocks B1, B2 such that block B2 can execute immediately after block B1. There is also an edge from every exception to every catch that might catch it. Figure 1 shows an example of a dataflow graph. Two special blocks are added to the CFG: ENTER and EXIT. There exists an edge from ENTER to every block B1 /∈ ENTER with no predecessor, and an edge from EXIT to every block B1 /∈ EXIT with no successor. A forward dataflow analysis propagates facts about the program from the ENTER to the EXIT node while a backward analysis propagates information backwards through the graph from the EXIT to the ENTER node.

Each node in the CFG is annotated with two sets: GEN and KILL. The KILL set contains facts in the analysis that become false in this node, and the GEN set contains facts that become true. Each node also has an IN and an OUT set. The sets associated with a block B can be denoted as IN[B], OUT[B], GEN[B], and KILL[B]. For a forward (backward) analysis the IN (OUT) set will contain facts that are true immediately before the node is visited, and the OUT (IN) set will contain facts that are true immediately after the node is visited. The transfer function describes how the OUT(IN) set is computed from the OUT(IN), KILL and GEN sets. For simplicity we only consider forward analysis from this point on. 

CFGs with branches contain join points where multiple paths come together, e.g., the block containing statements #8 and #9 in Figure 1. A meet operation decides how the values coming from the predecessor node are combined to form the value of the IN set. If the CFG contains cycles, an iterative algorithm that visit nodes repeatedly is used until the analysis converges to a fixed point such that revisiting all of the nodes does not change the values of any IN or OUT set. The algorithm works by adding the ENTER node to a work list. As nodes are processed, if their OUT set changes, their successors are added to the work list. When the work list is empty the algorithm has converged at a fixed-point. 

We note that all dataflow schemas compute approximations to the actual ground truth. The actual problem being solved is undecidable (e.g., constant propagation [5]). This is because it is undecidable, in general, if a particular path along a CFG will be taken during a program’s execution. As a result, dataflow solutions return conservative or safe estimates to the actual problem. A conservative approach guarantees that the results obtained by the analysis will err on the side of safety. Thus while an RD analysis may say more definitions reach some point than actually do, it will never fail to find all definitions that do reach a program point.

6.1.1 The Reaching Definitions Dataflow Problem

The first task in applying these concepts to the RD problem is to construct a CFG. Next, we define the GEN and KILL sets for each block B. The last assignment to a variable v in the block creates a definition d of v that can reach other statements outside the block, and therefore the definition is placed in the GEN set. Thus for block B1 of Figure 1 the definitions at c1, d2, and d3 can reach the other blocks, and therefore are added to B1’s GEN set. The definition of the KILL set comes from the following observation: an assignment to some variable v in a block prevents any definition of the variable v outside from flowing through the block. Thus any definition outside the block become members of the KILL set. Thus in block B5, definition d8 causes definition d4 to be in the kill set.

We now define the IN set. Consider the set of predecessors of some block B. Any definition that is in the OUT set of one of these predecessors can reach B, and thus is in IN[B] set of B. Therefore, IN[B] is the union of the OUT sets of all of its predecessors, e.g., IN[B2] is OUT[B5] ∪ OUT[B1]. Finally, the OUT set for a block B must be computed. The OUT set is simply the IN set with the effects of flowing through the block applied to it. The expression IN[B] - KILL[B] gives those definitions that reached the block and can reach later blocks, and unioning this with the GEN set gives all definitions that can pass through this block and reach other blocks. Thus fO : OUT[B] = GEN[B] ∪ (IN[B] - KILL[B]) is the transfer equation for block B.

Interprocedural analysis, which incorporates the effects of routine calls and routine arguments, is beyond the scope of this discussion but is covered in detail in [14].

6.2 No-Sleep Code Path Dataflow Analysis

We now formulate the single-thread no-sleep code path problem as an RD problem, and show how to solve it using standard dataflow analysis [6]. We only analyze non-reference counted, no-timer wakelocks and exotic component power APIs. We leave the study of other categories as future work.

6.2.1 No-Sleep Code Path to Reaching Definitions

For no-sleep code path analysis, we are only interested in the points in the code path where the smartphone component power is managed, e.g., the points in the CFG where wakelocks for the CPU or screen are acquired or released, or points where the camera is turned on and off. As a result, the domain of the dataflow problem is a set consisting of component wakelocks for traditional components and component power management assignments for exotic components. For brevity, from now on we use wakelocks to refer to the power control handles for both traditional and exotic components.

Once the transformation is completed, the no-sleep code path problem is reduced to finding the RD in the transformed CFG, i.e., finding which definition of a wakelock reaches the EXIT node of the CFG. If only those definitions that declare all the variables as 0 (i.e., the component can sleep) reach the EXIT node, the code is said to be free of no-sleep bugs, since all of the possible code paths put all accessed components to sleep before reaching the end of the CFG, and therefore the end of the code.

Solving the code path problem. We now show how to apply the standard iterative algorithm for dataflow analysis to solve our no-sleep code path problem.

For our no-sleep code path problem, the set of non-zero variable definitions reaching the EXIT node represents the no-sleep code path bugs in the app. Table 3 shows, for each block B, the IN[B] and OUT[B] sets at the end of three iterations. It shows the IN[B] and OUT[B] sets are the same at the end of the second and third
iteration, and hence the algorithm has reached a fixed-point and converged in three iterations. The value of \( OUT[EXIT] \) in the last iteration contains the reaching definitions at the end of the code: all definitions but \( d_4 \) and \( d_5 \) can reach the end of the code, including \( d_2, d_3 \) and \( d_6 \) which indicate the existence of a no-sleep bug. Their presence indicates no-sleep code paths along which a component (GPS, camera, and CPU wakelock_2, respectively) is woken up but not put to sleep.

6.2.2 Handling Uncaught Runtime Exceptions

Java runtime exceptions (RTE) \[45\] (e.g., null pointer and array index out of bounds exceptions) can be thrown during normal Java Virtual Machine (JVM) operations. RTEs that are handled explicitly by a try-catch block in code are handled as before by adding a path from the source block to the handler block. However, RTEs are often not handled by a program and the thread raising the exception is terminated by the JVM when the exception is thrown.

Uncaught RTEs are a source of no-sleep bugs and must be handled by our analysis. Consider the code in Listing 5. A CPU wakelock is acquired, followed by a call to the critical routine from instance \( b \). The wakelock is released after the call. If a RTE is raised \( \langle e.g. \rangle \), a null pointer exception on line 3 caused by \( b \) being null), the thread is halted. This results in a no-sleep energy bug since the thread terminates before the wakelock is released. We identify an instance of this bug in our characterization study. Listing 6 details the patch applied by the developer to fix a null pointer RTE (code lines appended with "-" or "+" indicate that these lines were removed from or added to the new version, respectively). The developer added handlers for the null pointer RTE and moved the lock release into a finally block to ensure that it is run regardless of any exceptions.

Our technique places an edge from each RTE that is not handled within a routine to the \( EXIT \) node for that routine. This creates a path for a lock acquire definition to reach the exit, and could lead to more false positives (although we have not seen that in our test cases). Techniques such as null pointer analysis \[46\], ABCD \[47\] for array bounds check, and RTE analysis techniques, e.g., \[48\], can be used to make the analysis more precise and generate fewer false positives.

6.2.3 Handling Event Based Entry Points

Android app programming is primarily event-based programming. Unlike traditional code, where the \( main() \) routine starts the app with the app exiting when \( main() \) returns, Android app programming typically consists of several functions which are event handlers, one corresponding to each event the app handles. These events could be a button click, an incoming call, a notification from server, a response from a component \( \langle e.g. \rangle \), GPS, etc. Each event handler is invoked when the event is fired and the handler may in turn invoke a tree of routines underneath it before exiting.

Handling multiple entry points of an app creates a new challenge: each handler has its own CFG \[49\] and a component may be turned on in one event handler and put to sleep under another \( \langle e.g. \rangle \), start camera when start button is clicked and stop camera when stop button is clicked). However, the order of execution of the different events, which is needed to stitch together the CFGs of different handlers, may be unknown at compile time and depends on user interactions.

We handle this complication as follows. (1) For common event handlers \( \langle e.g. \rangle \), onCreate, onPause which have known invocation orders, we simply perform the RD analysis across them on the combined CFG obtained from stitching together individual CFGs following those invocation orders. For example, if a component is not put to sleep when the app is paused after being first created, it usually is a sign of a no-sleep bug. (2) For the remaining handlers, we ask the developers to specify all expected invocation orders, and then perform no-sleep bug RD analysis on the combined CFG for these orders.

7. NO-SLEEP RACE CONDITIONS

To statically detect possible no-sleep race conditions for multi-thread apps, we adapt the RD dataflow analysis previously developed for parallel programs \[49,50\].

A multi-threaded program typically has a repeating pattern of sequential sections ending with a thread fork, interleaved execution of parallel threads followed by a thread join, followed by the next sequential section in the pattern. Execution is sequential within each thread and so a CFG can be built for the thread. CFGs for different threads can be stitched together by connecting the fork spawning the thread with the ENTRY node for the thread’s CFG, and the EXIT node of the CFG with the join node using a parallel control edge \[50\].

The RD analysis is now modified for this new CFG. Three observations \[50\] motivate these modifications. (1) All threads in a parallel section are executed; (2) Any of the definitions \( d_1, d_2, \ldots, d_k \) to some variable \( v \) executing in different threads, and not ordered

\[ \text{Listing 6: Fixing no-sleep code path due to runtime exceptions.} \]

```java
wake_lock_.acquire(); //CPU should not sleep
Object b = xyz.getObject(); //b is a reference to an object
b.net_sync(); //Perform critical task here

+ try {
  + client = new AppengineClient(this);
  + log.d(TAG, "onHandleIntent");
  +
  + finally {
    + wake_lock_.release();
  
```
by synchronization, may be the last definition of \( v \) to execute, therefore no such definition \( l_{pq} \) can be said to kill another definition \( d_{ij} \), \( p, q \in \{i, j, \ldots, k\} \); (3) Any kill performed unconditionally by any thread in a parallel section (e.g., along all sequential paths through the thread) kills all definitions that occur before the parallel section’s fork; (4) Any definition \( d \) performed conditionally within a thread does not kill definitions \( d' \) before the thread’s parallel section’s fork since either \( d \) or \( d' \) may reach the parallel section’s join, and following statements. The details of modifying the dataflow analysis to account for these additional constraints and ordering synchronization are discussed in detail in \[5] and are not repeated here.

Figure 2 shows that the RD analysis on the PFG finds that definitions \( d_1, d_4 \) and \( d_7 \) can reach the EXIT node. Since \( d_7 \) turns on the component, this is a no-sleep race bug.

8. IMPLEMENTATION

ProGuard Extension: We implemented no-sleep bug tracing as a 1K-LOC extension to ProGuard \[18\]. The ProGuard tool is used to shrink, optimize and obfuscate Android code and helps to make smaller .apk installer files. It builds an intermediate Representation of the input source containing CFGs that we use. We chose ProGuard since it is integrated into the Android build system, it automatically runs when an Android app or the framework is compiled and does not require a separate, manual invocation. However, source code is not required to perform the analysis since ProGuard can run directly on the bytecode generated by the Java compiler. If we only have the .apk installer for an app we first use \texttt{dex} \[16\] to decompile the embedded .dex files (Dalvik Executable \[31\]) and convert them to Java bytecode (.class files). We then run ProGuard and the no-sleep bug dataflow analysis on the .class bytecodes.

Handling Object References and Intent Resolution: Java object references and intent resolutions are the only indirect control transfer mechanisms in the Android framework and apps. An indirect control transfer mechanism poses problems for static analysis since it is difficult to determine at compile time which class the object is an instance of, or which handler will service the intent, and hence which particular method (routine) will be called. We use a conservative approach by analyzing all routines’ references that could possibly be referred to at runtime.

Handling Special Code Paths: To reduce the number of false positives, we handle the two special cases shown in Listing 7. Two wakelocks (\texttt{wl1}, \texttt{wl2}) are acquired before if conditions and are released under their respective if. We found these two usages to be common in mobile apps. The RD no-sleep bug analysis for code in Listing 7 would flag both lock acquires as reaching the end of the block since they are not released in the else branches. However, in both cases, it is evident that there is no bug since if the wakelock is either null or not held, it need not be released. We handle these two common usages specially, by inserting an else branch to the if condition which contains a definition of release().

Runtime Exceptions: In our characterization study, we did not observe any occurrences of uncaught RTEs other than null pointer exceptions (NPEs). Hence, in our current implementation, we only handle NPEs. We leave handling other RTEs as future work. To handle NPEs, we trace the null reaching definitions at each access point of the object for every object declared in the program. If a null definition reaches an object access point, we add a path from that point to the EXIT node in the CFG.

Race Conditions: Our current implementation only implements analysis for no-sleep race condition for programs without synchronization points. We leave as future work extensions to handle synchronization points, which can be implemented by adopting the techniques proposed in \[5\].

9. EVALUATION

We now present experimental results of no-sleep bug detection using our dataflow analysis based no-sleep bug detection tool. We first present a summary of the detection results on 500 apps running on Android and then discuss false positives and the runtime of the scheme.

Methodology: We collected app installers (.apk files) for 500 apps, including popular apps like Facebook, Google apps such as gtalk and stock apps in the Android framework including Email and Dialer. These include all the apps listed in Table 2. Automatic analysis of the manifest.xml file for permissions reveals that 187 apps explicitly manipulate component wake/sleep cycles. We then decompiled the .apk installers using \texttt{dex} \[16\] and obtained

<table>
<thead>
<tr>
<th>Block B</th>
<th>OUT(B)</th>
<th>IN(B)</th>
<th>OUT(B)</th>
<th>IN(B)</th>
<th>OUT(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3}</td>
<td>{d1,d2,d3}</td>
<td>{d1,d2,d3}</td>
</tr>
<tr>
<td>B2</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B3</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B4</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B5</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B6</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B7</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
</tbody>
</table>

Table 3: Computing IN and OUT for no-sleep code paths.

<table>
<thead>
<tr>
<th>Block B</th>
<th>OUT(B)</th>
<th>IN(B)</th>
<th>OUT(B)</th>
<th>IN(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3}</td>
<td>{d1,d2,d3}</td>
</tr>
<tr>
<td>B2</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B3</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B4</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B5</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B6</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
<tr>
<td>B7</td>
<td>{</td>
<td>{</td>
<td>{d1,d2,d3,d4,d5}</td>
<td>{d1,d2,d3,d4,d5}</td>
</tr>
</tbody>
</table>

Listing 7: Handling simple code paths.

1. \texttt{wl1.acquire(); \texttt{wl2.acquire();} //wakeup
2. if(wl1 != null) \texttt{if object is not null
3. \texttt{w1.release(); //release the wakelock
4. \texttt{if(wl2.isHeld()) \texttt{if \texttt{wl2 is acquired
5. \texttt{wl2.release(); \texttt{release the wakelock

Figure 2: Tracing no-sleep race bug: Parallel Flow Graph (PFG) for code in Listing 4.

We implemented no-sleep bug tracing as a 1K-LOC extension to ProGuard \[18\]. The ProGuard tool is used to shrink, optimize and obfuscate Android code and helps to make smaller .apk installer files. It builds an intermediate Representation of the input source containing CFGs that we use. We chose ProGuard since it is integrated into the Android build system, it automatically runs when an Android app or the framework is compiled and does not require a separate, manual invocation. However, source code is not required to perform the analysis since ProGuard can run directly on the bytecode generated by the Java compiler. If we only have the .apk installer for an app we first use \texttt{dex} \[16\] to decompile the embedded .dex files (Dalvik Executable \[31\]) and convert them to Java bytecode (.class files). We then run ProGuard and the no-sleep bug dataflow analysis on the .class bytecodes.

Handling Object References and Intent Resolution: Java object references and intent resolutions are the only indirect control transfer mechanisms in the Android framework and apps. An indirect control transfer mechanism poses problems for static analysis since it is difficult to determine at compile time which class the object is an instance of, or which handler will service the intent, and hence which particular method (routine) will be called. We use a conservative approach by analyzing all routines’ references that could possibly be referred to at runtime.

Handling Special Code Paths: To reduce the number of false positives, we handle the two special cases shown in Listing 7. Two wakelocks (\texttt{wl1}, \texttt{wl2}) are acquired before if conditions and are released under their respective if. We found these two usages to be common in mobile apps. The RD no-sleep bug analysis for code in Listing 7 would flag both lock acquires as reaching the end of the block since they are not released in the else branches. However, in both cases, it is evident that there is no bug since if the wakelock is either null or not held, it need not be released. We handle these two common usages specially, by inserting an else branch to the if condition which contains a definition of release().

Runtime Exceptions: In our characterization study, we did not observe any occurrences of uncaught RTEs other than null pointer exceptions (NPEs). Hence, in our current implementation, we only handle NPEs. We leave handling other RTEs as future work. To handle NPEs, we trace the null reaching definitions at each access point of the object for every object declared in the program. If a null definition reaches an object access point, we add a path from that point to the EXIT node in the CFG.

Race Conditions: Our current implementation only implements analysis for no-sleep race condition for programs without synchronization points. We leave as future work extensions to handle synchronization points, which can be implemented by adopting the techniques proposed in \[5\].
Table 4: Summary of detecting no-sleep code paths.

<table>
<thead>
<tr>
<th>App type breakdown</th>
<th>#</th>
<th>Breakdown of 42 apps that contain no-sleep code paths</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total input set of apps</td>
<td>500</td>
<td>In the framework</td>
<td>30</td>
</tr>
<tr>
<td>Manipulated component</td>
<td>187</td>
<td>Incorrect event handling</td>
<td>6</td>
</tr>
<tr>
<td>Fully decompiled</td>
<td>86</td>
<td>if, else + exception paths</td>
<td>26</td>
</tr>
<tr>
<td>No-sleep code paths</td>
<td>42</td>
<td>Forget release (incl. Services)</td>
<td>12</td>
</tr>
<tr>
<td>False positives</td>
<td>13</td>
<td>Miscellaneous</td>
<td>3</td>
</tr>
<tr>
<td>True negatives</td>
<td>31</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

86 apps that were decompiled\(^{10}\) to bytecode and Java source code. For these 86 apps, we ran our detection analysis tool directly on the bytecode, and then used the decompiled Java source code to identify false positives incurred by our analysis. Table\(^{3}\) gives a breakdown of the apps (left) and causes of bugs in them (right).

**No-Sleep Bug Analysis:** Manually enumerating all possible paths in the program to verify the correctness of the tool along each of those paths is not humanly possible due to the exponential number of paths. We used the following approach. For each of the 86 apps, we manually\(^{11}\) the no-sleep bugs (no-sleep code paths in single threads). We then check if the bugs had been reported by the tool. Using this approach we segregated the apps into four categories: (a) **True Positive (TP):** Apps in which we manually found the bug and it was reported by the tool; (b) **True Negative (TN):** Apps in which we could not manually find bugs and the tool reported no bug; (c) **False Positive (FP):** Apps in which we did not find bugs manually, but the tool reported bugs; and (d) **False Negative (FN):** Apps where we found a bug manually but the tool did not. The last category contained no apps. Table\(^{4}\) summarizes the results. We found 31 apps in the TN set, 13 apps in the FP set and 42 apps in TP set. Below we break down these 42 apps according to the causes.

1. **Incorrect event handling** (26): The largest category of bugs in these apps are bugs from inappropriate handling of events in Android apps, specifically the handlers in the default Android activity cycle\(^{52}\): `onPause()` and `onStop()`. An activity is a single focused window that a user interacts with. It is the foreground GUI part of an app that the user sees. The framework calls the `onPause()` event routine of the activity in the app whenever the activity is interrupted by another activity coming in front of it, e.g., an incoming phone call displaying an “incoming call-box” while the user is playing a game. Once the activity is completely sent to the background, i.e., the activity is no longer visible, the framework calls `onDestroy()` when the app finally exits.

   The bug occurred in any app that wakes up a component when it is started or resumed in the default event handlers `onCreate()` or `onResume()`, respectively, and lets the component sleep only when the app finally exits, in `onDestroy()`. Basically, when such an app is interrupted during this interval by another activity started by the user (like clicking the home screen, or starting another app), the respective component is kept awake since the original app is frozen and may not run for a long time. We observed instances of this bug in 26 apps involving CPU wakelocks (in 23) and GPS (in 3).

2. **if-else + exceptions** (12): In 12 apps, including Agenda Widget and Android Email App (Table\(^{2}\), not all paths (using if-else) released the component. Also, no-sleep code paths

\(^{10}\)Decompilation failed completely for the remaining apps, i.e., not even a single source file was generated.

\(^{11}\)These include leveraging bug reports when available for apps in Table\(^{2}\).

Listing 8: No-sleep code path: false positives.

```java
// Use a routine to manipulate component
void WakeUpCPU(boolean wakeup)
{
    if(wakeup) wl.acquire(); // wake up
    else wl.release(); // release the lock
} // End WakeUpCPU

void CriticalTask()
{
    WakeUpCPU(true); // acquire the lock
    // Do critical task ...
    WakeUpCPU(false); // release the lock
} // End CriticalTask
```

Akin to the code in Listing\(^{8}\) were observed in apps including the Facebook and Android Email apps. The component (CPU in these cases) was not allowed to sleep in case of a thrown exception.

3. **Forgot to release a wakelock** (3): In K9mail \(^{43}\), Agenda Widget \(^{21}\) and SIP service \(^{12}\), the programmer forgot to release the CPU wakelock.

4. **Miscellaneous** (1): In Android WifiService \(^{31}\), the wakelock is not released in all cases of received messages.

We also detected a no-sleep bug caused by a race condition in the Android Email app, as was discussed in detail in §4.2. Analyzing multi-threaded Android apps involving explicit synchronizations is left as future work.

**Reasons for False Positives:** 13 apps were reported to contain a no-sleep code path, but upon further manual analysis, they turned out to be false positives, as discussed next.

There were two major reasons for the false positives reported in the 13 apps: use of helper functions or variables for component access, and interference of higher level app logic in lock placement.

(a) **Helper functions/variables** (3): The code in Listing\(^{8}\) describes an app using a helper function for wakelock manipulation. Routine `WakeUpCPU(boolean)` manipulates the wakelock depending on the boolean input variable. Routine `CriticalTask()` acquires and releases the wakelock using the helper routine by passing `true` and `false` as input arguments. RD analysis of routine `WakeUpCPU(boolean)` suggests that one code path (the `if` branch of the condition) reaches the end of the routine and hence there exists one path in routine `CriticalTask()` that reaches the end of the routine with the wakelock held.

Similarly, we found apps to use additional helper variables to track the liveness status of the component instead of relying on standard API routines (e.g., `isHeld()`) that come bundled with the Android framework. The boolean helper variable is toggled each time the component is switched on or off and is checked before setting the component free. This use of multiple variables to achieve a single purpose not only results in false positives in our static analysis of no-sleep bugs, but also was previously found to be the root cause of multi-variable access bugs\(^{17}\).

(b) **Higher level App logic** (10): Listing\(^{8}\) demonstrates a false positive observed in the Android Dialer app due to higher level app logic. The incoming call handler acquires the wakelock if the caller is not blacklisted by the user and otherwise it immediately returns. Similarly, when the call is disconnected, the app releases the wakelock only when the caller was not blacklisted. This induces a false positive in static reaching definitions analysis since the `acquire()` in `HandleIncomingCall()` reaches the end of `disconnectCall()`. We leave addressing this issue to future work.

**Analysis Runtime and Wakelock Statistics:** Table\(^{5}\) presents detailed statistics of no-sleep bugs in 5 popular apps. For each app, the table shows the LOC, the number of classes in the app and in
the libraries, the number of wakelocks, and how many times they were defined, i.e., acquired and released, and the time taken to run the no-sleep code path analysis.

From the table we see that the runtime of the analysis varied from 3 seconds (for CommonsWare (CW), 0.3 KLOC) to 408 seconds for FaceBook (93.5 KLOC decompiled). The 3 services that belong to the Android framework took about 50 seconds to perform the code analysis. Since the tool performs off-line static analysis, i.e., on a desktop/server, there is no energy drain on the mobile devices. The table also shows that the Telephony service in Android holds 7 different wakelocks (all with option PARTIAL_WAKE_LOCK) with 18 acquire and 29 release definitions, for different utilities, including CDMA connection, GSM connection, SMS Dispatcher, Radio Interface Layer, etc. Other apps utilize one or two wakelocks. Although the FaceBook app manipulates only one wakelock, the wakelock APIs could be called from multiple locations, resulting in a total of 256 acquire and 128 release definitions.

Battery Drain of No-Sleep Bugs: Table 2 lists the amount of time it will take to drain a fully charged battery under typical usage on the Google Nexus One phone due to the corresponding no-sleep bugs.

Real App CFG (Android Email App): To illustrate the complexity of the apps, apart from the number of lines of code and number of wakelock objects shown in Table 2, we show in Figure 3 a trimmed down version of the CFG of one of the central routines in the Android Email app, runPingLoop(). The nodes in the graph show branch points. These include, but are not limited to, handling error conditions such as (a) PING returned empty, (b) PING returned with a login failure, (c) NAT failure, (d) synchronization aborted by the user, and (e) alarm event trigger. The left-most node (solid black) denotes the entry point and the right-most node denotes the exit point. The edges depicted in thick black lines are the ones along which the CPU wakelock is being held, while edges in dotted green lines are when wakelocks have been released. Out of the four edges reaching the exit node, only one reaches after releasing the wakelock.

![Figure 3: CFG of a routine in the Android Email app.](image)

### 10. RELATED WORK

**Debugging in Mobile Environments:** Diagnosing bugs in smartphone environment is a new domain, and there is little existing work. MobiBug [53] is a framework for mobile debugging that focuses on how to perform lightweight logging on resource-limited smartphones. It proposes three ideas: spreading the logging task among many phones, building a conditional distribution model for the app behavior and its dependencies, and sampling what each phone logs. MobiBug is designed to be a runtime traditional bug tracing system, targeting bugs that usually result in app crashes. However, energy bugs [4] differ from traditional bugs in that they do not lead to any app crash; the apps continue to work normally except that the battery drains rapidly. Figure 4 presents a taxonomy of many types of energy bugs in smartphones and estimates no-sleep bugs to constitute about 70% of all energy bugs in smartphone apps. Our paper is the first directed towards solving energy bugs in mobile phones.

**Applications of Reaching Definitions:** Reaching definitions analysis has many uses, and most are part of the compiling folklore (see [3]).

**Debugging Software Concurrency Bugs:** Debugging traditional software bugs on non-mobile devices is a well-studied topic. There have been several bug characterizing studies (e.g., [5, 54]) which classify different categories of software bugs, e.g., concurrency bugs, semantic bugs, and configuration bugs. The debugging solutions fall into three categories: static, dynamic, or hybrid analysis. Some examples of static analysis include Racex [55] which uses a flow sensitive, inter-procedural analysis to detect data races and deadlocks, and MUVI [7] which applies data mining to infer patterns of multi-variable access correlations. Our solution for no-sleep energy bugs falls in the realm of static analysis, and is based on reaching definitions analysis.

### 11. CONCLUSION

This paper makes the first advances towards understanding and automatically detecting software energy bugs on smartphones. It makes three contributions. First, it presents the first comprehensive real world no-sleep energy bug characterization study. Our study reveals three major causes of no-sleep energy bugs and provides useful guidelines and hints to design effective detection schemes. Second, it proposes the first detection solution, based on the classic reaching definitions dataflow analysis, to automatically infer potential no-sleep bugs in an app. Third, evaluation of our tool on 86 Android apps and the Android framework shows that our tool accurately detected all reported instances of no-sleep bugs, as well as 30 instances of new no-sleep bugs.

Our work is only the beginning of smartphone energy bug research and opens a wide avenue for further studies. A sample list of future work includes: (1) develop solutions to no-sleep dilation bugs; (2) explore run-time and hybrid compile-time and run-time solutions to no-sleep bugs; (3) characterize and detect other types of energy bugs in smartphone apps; (4) understand energy bugs in the smartphone OSes; and (5) develop better programming language support to avoid no-sleep bugs at programming time.
Acknowledgments. We thank the anonymous reviewers, especially our shepherd, Maria Ebling, for their helpful comments. This work was supported in part by NSF grant 0916901. Abhinav Pathak was supported in part by a 2011 Intel PhD Fellowship.

12. REFERENCES

[1] “Smartphone sales overtake pcs for the first time.” URL: http:// mashable.com/2012/02/03/smartphone-sales-overtake-pcs/


[19] “Sipservice: release wake lock for cancelled tasks.” URL: https://github.com/android/platform_frameworks_base/commit/0c016e6060d079b025a44c1159b63944ace17


[22] “Facebook 1.3 not releasing partial wake lock” URL: http://geekfor.me/news/facebook-1-3-wakelock/

[23] “K9mail: Simplifying wakelock.” URL: http://code.google.com/p/k9mail/source/detail?r=1696

[24] “Checkinmap: Disable location updates when checkinmap is disabled.” URL: https://github.com/mjsanchek/Ushahidi_Android/commit/3374b48f5f272f51e84796fab129471f6e3c0357


[26] “Babblesink: Move line inside try in case of npe before release of wake lock.” URL: https://github.com/hatstand/babblesink/commit/91bc6f01e81e1f6425a5b62a34b787fe6080e36

[27] “Ensuring that the wakelock is released during exception.” URL: https://github.com/commonsjoy/cwac-wakelock/commit/c74440f1150887bb9a1a3c44015c759fd7ab1970

[28] “frameworks/base/telephony: Release wakelock onril request send error.” URL: https://gist.github.com/ CyanogenMod/android_frameworks_base/commit/135122d7/aa88a8e4095e6c128984d1d0b171b1b

[29] “Ensure wakelock is released when an IOException occurs during a sync.” URL: https://github.com/muton/android_apps_email/commit/85f6ce473c444136c86ed4f092c3d3be925e23e733

[30] “Android issue #9307 fixed - partial wakelock release.” URL: https://github.com/CyanogenMod/android_packages_apps_Email/commit/f53bf8f1783808d882a034e10c419c82f4b93

[31] “Wakelock issue for driver stop.” URL: https://github.com/muton/android_apps_email/commit/85f6ce473c444136c86ed4f092c3d3be925e23e733

[32] “Fix wakelock leak in powermanagerwakelock.sendnotificationlocked().” URL: http://gitorious.org/rowboat/frameworks-base/commit/53b8f8f1783808d882a034e10c419c82f4b93

[33] “Using a locationlistener is generally unsafe for leaving a permanent partial_wake_lock.” URL: http://code.google.com/p/android/issues/detail?id=4533

[34] “Locationmanagerservice: Fix race when removing locationmanagerservice.” URL: https://gist.github.com/ CyanogenMod/android_frameworks_base/commit/0528b9b26a9d64ba43acd0e334638303d514b8eb#location/android/location/ILocationProvider.aidl


[36] “E-mail app has a bug which causes a partial wake lock to be held until manually interrupted.” URL: http://code.google.com/p/android/issues/detail?id=6811


[38] “Googlebackuptransport holds backup wakelock so long which leads to high current.” URL: http://www.google.bg/support/forum/p/Google+Mobile/thread?id=481ff31338a19536

[39] “Fix threading problem that resulted in the wakelock being held too long.” URL: https://github.com/CyanogenMod/android_frameworks_base/commit/a6b2c8351926289227af1a/334638303d514b8eb#location/android/location/ILocationProvider.aidl

[40] “Googlemaps holding wakelock for long.” URL: http://www.google.com/support/forum/p/maps/thread?id=6901f13338a19536

[41] “Java.lang.Exception class.” URL: http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Exception.html
“Email 2.3 app keeps awake when no data connection is available.” URL: http://www.google.com/support/forum/p/Google+Mobile/thread?tid=53be134321358e8


“java.lang class runtimeexception.” URL: http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/RuntimeException.html


M. Bravenboer and Y. Smaragdakis, “Exception analysis and points-to analysis: better together,” in International symposium on Software testing and analysis, 2009, pp. 1–12.


“.dex: Dalvik executable format.” URL: http://source.android.com/tech/dalvik/dex-format.html


S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl, “There’s an app for that, but it doesn’t work. diagnosing mobile applications in the wild,” in Hotnets, 2010.
