Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs

Panagiotis Vekris
Ranjit Jhala, Sorin Lerner, Yuvraj Agarwal

University of California, San Diego
Software Energy Bugs
Software Energy Bugs

News: Facebook 1.3 Not Releasing Partial Wake Lock

GeekForMe is long overdue for a post explaining awake time and when you should and shouldn’t worry about it, and I actually worry that this post could bring that whole debate back up, but I wanted to get this information out there.

The official Facebook application for Android was just updated to version 1.3. Soon after there were many reports of people’s phones not sleeping, including [this one](http://www.fresh-rom.org/) and [this one](http://www.geekforme.com/) just as examples. I’m not a big Facebook user but I do have experience in tracking down phone sleeping issues so I decided to take a quick look. I will mention in my testing I did not have Facebook syncing my contacts but that’s the only setting I changed from the default setup.

First I’ll give a quick explanation on what a wake lock is. Certain applications may find a reason to need to keep your phone awake. This is most common with something like a game so that your screen doesn’t turn off while you are in the middle of playing it. There are actually 4 different types of wake locks. The bottom 3 in the chart will all keep your screen on but will still allow your

<table>
<thead>
<tr>
<th>Flag Value</th>
<th>CPU</th>
<th>Screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL_WAKE_LOCK</td>
<td>On*</td>
<td>Off</td>
</tr>
</tbody>
</table>
Power Manager

Concept

*Opportunistic suspend*

WakeLock API

*App developer directives*

*Keep a device component awake*
WakeLock Use

Lock Based Mechanism

```java
PowerManager pm = getSystemService(POWER);
WakeLock l = pm.newWakeLock(SCREEN);
l.acquire();
//... screen stays on ...
l.release();
```
WakeLock Use

Lock Based Mechanism

PowerManager pm = getSystemService(POWER);
WakeLock l = pm.newWakeLock( SCREEN );
l.acquire();
//... screen stays on ...

“No-sleep” bugs

*Misuse of WakeLock API*

*Screen Awake => 25% battery drain per hour [Pathak’12]*
Static Analysis

Track No-Sleep Bugs

Pathak [MobiSys’12]

Reaching definitions dataflow analysis
Detect known instances & find new ones

Our Work

Define policies => Verify absence of bug
Track asynchronous calls
Precise dataflow analysis
Activity

A screen to interact with

Application Component

Building block

Entry point

Stack Model

Top of stack = visible activity
Activity States

Callbacks

Called by system on state transitions
onCreate() {
    super.onCreate();
    // setting up
    l = newWakeLock(SCREEN);
}

onForeground()
onCreate() {
    super.onCreate();
    // setting up
    l = newWakeLock( SCREEN );
}

onForeground() {
    // ...
    l.acquire();
}

onBackground()
onCreate() {
    super.onCreate();
    // setting up
    l = new WakeLock( SCREEN );
}

onForeground() {
    // ...
    l.acquire();
}

onBackground()
onCreate() {
    super.onCreate();
    // setting up
    l = newWakeLock( SCREEN );
}

onForeground() {
    // ...
    l.acquire();
}

onBackground() {
    // ...
    l.release();
}
onCreate() {
  super.onCreate();
  // setting up
  l = newWakeLock( SCREEN );
}

onForeground() {
  // ...
  l.acquire();
}

onBackground() {
  // ...
  l.release();
}
onCreate() {
    super.onCreate();
    // setting up
    l = newWakeLock(CPU);
    l.acquire();
}

// unimplemented
onForeground();

// unimplemented
onBackground();
onCreate() {
    super.onCreate();
    // setting up
    l = newWakeLock(CPU);
    l.acquire();
}

// unimplemented
onForeground();

// unimplemented
onBackground();

Application in background holds CPU WakeLock

Acquire CPU WakeLock

iZen Lite V2.3
onCreate() {
    super.onCreate();
    // setting up
    l = new WakeLock(CPU);
    l.acquire();
}

// unimplemented
onForeground();

// unimplemented
onBackground();

Application in background holds CPU WakeLock without doing useful work

Incorrect Use

iZen Lite V2.3

Acquire CPU WakeLock
Invisible Activities should not hold resources

Activity.onBackground( ) should release all WakeLocks
Invisible Activities should not hold resources

Activity.onBackground() should release all WakeLocks
Invisible Activities should not hold resources

Activity.onBackground() should release all WakeLocks
Invisible Activities should not hold resources

Activity.onBackground() should release all WakeLocks

Expected State
Policies

WakeLocks

*Used in any component - Not only Activities*
*Their effects cross component boundaries*

General policies

We defined policies for basic components
Conservative for the rest
Static Analysis to enforce them
Tool Overview

Uses a precise Dataflow algorithm [RHS’ 95]

State: Set of WakeLocks that

*may be held*

*at given program point*

Validate

*Check if state complies to policies*
How our tool works

```java
onForeground() {
    l = newWakeLock(CPU);
    if (random() % 2 == 0)
        l.acquire();
}

onBackground() {
    if (random() % 2 == 0)
        l.release();
}
```
State: `{}`

```java
l = newWakeLock(CPU)
{}
{}
random() % 2 == 0
{}
{}
l.acquire();
random() % 2 == 0
{}
{}
l.release();
```

WakeLocks that *may* be held.
Initially empty set
l = newWakeLock(CPU)

random() % 2 == 0

l.acquire();

random() % 2 == 0

l.release();

1 is associated with CPU

1 -> CPU

State: {}
State: {}  

l = newWakeLock(CPU)  

random() % 2 == 0  

l.acquire();  

random() % 2 == 0  

l.release();  

State is propagated to both branches  

Annotation: 1 -> CPU
1 = newWakeLock(CPU)

1.acquire();

1.release();
l = newWakeLock(CPU)

random() % 2 == 0

l.acquire();

random() % 2 == 0

l.release();

Performed on WakeLock object l

Virtual method call to acquire

l.acquire();

Annotation: l -> CPU
l = newWakeLock(CPU)

random() % 2 == 0

l.acquire();

random() % 2 == 0

l.release();

Annotation: l -> CPU
State:  

```java
l = newWakeLock(CPU)

{}  

random() % 2 == 0

{}  

{}  

l.acquire();

{}  

{}  

random() % 2 == 0

{}  

l.release();
```

Annotation:  

1 -> CPU

Acquire introduces l to set
State: {}

1 = newWakeLock(CPU)

random() % 2 == 0

1.acquire();

random() % 2 == 0

1.release();

Annotation: 1 -> CPU

Join Statement

A U B
State: `{}`

```java
l = newWakeLock(CPU)

random() % 2 == 0

l.acquire();

random() % 2 == 0

{l}

l.release();

Union of input sets
```

Annotation: 1 -> CPU
State: {}

```java
l = newWakeLock(CPU)
{}
random() % 2 == 0
{}
{}
l.acquire();
{} {1}
random() % 2 == 0
{1} {1}
l.release();
{} {1}
```

Annotation:

A

1 -> CPU

A-{} 1
State: `{}`

1 = newWakeLock(CPU)

random() % 2 == 0

Release removes 1 from set

Union of input sets

Annotation: 1 -> CPU
WakeLock 1 may be held at the end of the code
l.acquire();

// ... critical operation

if (l.isHeld())
    l.release();

// ... Wakelock released
Path Sensitivity

1.acquire();

l.isHeld();

true

false

l.release();

1 is not held

Remove it from set

State empty in the end
Context Sensitive Method Calls

myRelease() {
    if (l.isHeld())
        l.release();
    return;
}

If l is held in the before calling myRelease
It will be released after

Method Summary
myRelease();

{ l }

if (l.isHeld())
    l.release();
return;

myRelease()
{ }
Handling Asynchrony

Intent calls

Asynchronous messages among components

Threaded execution

Java Threads

Posted on Message Queue (Android)
Handling Asynchrony

Component A

Component B

start(B);
Handling Asynchrony

Component A

l.acquire();
{1}
start(B);

Component B

l.release();
{1}

WakeLock set is preserved at component call

Empty state is propagated back to caller component
Handling Asynchrony

Component A

l.acquire();

start(B);

Component B

l.release();

So does Component B

Component A exits in released state
Handling Asynchrony

Component A

```java
l.acquire();
start(B);
l.release();
```

Component B

Component B does not operate on WakeLock l
Handling Asynchrony

Component A

```java
l.acquire();
start(B);
l.release();
```

Component B

```java
{1}
```

Component B does not operate on `WakeLock`

Policies will not be checked against it

Component A exits with released state
Handling Asynchrony

Component

{ } Start();

{l<async>} l.acquire();

{l<async>} l.release();

Helper Thread

{ } l.acquire();

{1}

Race Condition

Release does not remove asynchronously acquired lock
Handling Asynchrony

Component

```
Start();
```

`{ }`

Helper Thread

```
l.acquire();
{1}
```

`{ }`

```
l.release();
```

Race Condition

Policy violated
Handling Asynchrony

Component

{ }
Start();
l.release();
{l<async>}

Helper Thread

{ }
l.acquire();
{l}

Race Condition
Tool Workflow

Apk (Dalvik Bytecode ≠ Java Bytecode)
- 740 apps with wakelocks (out of 2718)

Retargeted Java Bytecode

Input to WALA/our tool

Optimized Java Bytecode
- 328 correctly decompiled

Ded/Dex2jar

Soot
Analysis Results for 328 apps

- **No Policy Violations**: 44%
- **Policy Violations**: 56%

- Many bugs in real apps
- Too imprecise analysis
- Manual inspection of 31 violating apps
True Positives

Apps confirmed to contain a bug: 14 of 31

- Lifecycle Abuse: 10
- Incorrect release conditions: 2
- Not all paths released: 2
False Positives

Correct apps flagged as buggy: 17 of 31

- Complex release conditions: 7
- Too strict policies: 4
- Unresolved Intent calls: 3
- Wrapper with parameter: 2
- Unrecognized callback: 1
Soundness

WALA

Conservative at handling exception edges

```
Foo() {
    ... a.f;
    ... return;
}
```

Keep analysis precise:

*Exception edges bypass release operations*

Kill state over exception edges
Future work

Soundness

Exception edges

Precision

Add path sensitivity
Asynchronous calls $\rightarrow$ precise points-to analysis
Refine policies

Retarget Dalvik effectively
Conclusion

No-sleep bugs are real

We presented a tool that:

* Uses static analysis
* Guarantees for absence of no-sleep bugs
* Tracks asynchronous calls