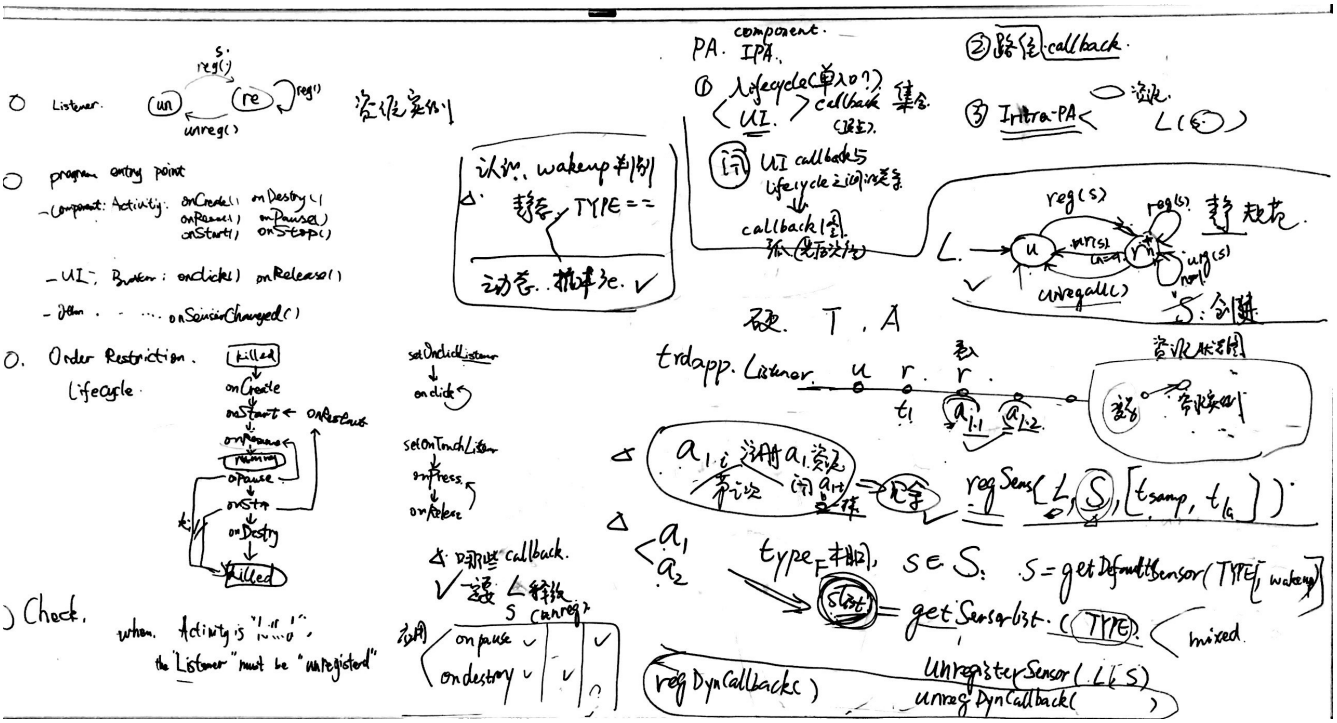
Contents

1 Overview



Part I
Android Platform

This part is mainly about the survey of the Android Platform

2 Offical Documents for Sensors

2.1 SDK referecnce (for developers)

Sensor Overview

- Motion Sensors

  The accelerometer and gyroscope sensors are always hardware-based.

  The gravity, linear acceleration, rotation vector, significant motion, step counter, and step detector sensors are either hardware-based or software-based.

- Position Sensors

  geomegnetic field sensor, orientation sensor and proximity sensor.

- Environment Sensors

  ambient temperature, light, pressure, relative humidity and temperature sensors.

Important Classes:

SensorManager You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

Sensor You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

SensorEvent The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

SensorEventListener You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

note:

- **Unregister sensor listeners** from document: Sensors Overview

  Always make sure to disable sensors you don't need especially when your activity is paused. Failling to do so can drain the battery in just a few hours. Note that the system will not desable sensors automatically when the screen turns off.

  It's also important to note that this example uses the onResume() and onPause() callback methods to register and unregister the sensor event listener. As a best practice you should always disable sensors you don't need, especially when your activity is paused. Failing to do so can drain the battery in just a few hours because some sensors have substantial power requirements and can use up battery power quickly. The system will not disable sensors automatically when the screen turns off.

- **Don't block the onSensorChanged() method** from document: Sensors Overview

  Sensor data can change at a high rate, which means the system may call the onSensorChanged(SensorEvent) method quite often. As a best practice, you should do as little as possible within the onSensorChanged(SensorEvent) method so you don't block it. If your application requires you to do any data filtering or reduction of sensor data, you should perform that work outside of the onSensorChanged(SensorEvent) method.

- **Wakelock for non-wake-up sensors** from document: SensorManager

  In the case of non-wake-up sensors, the events are only delivered while the Application Processor (AP) is not in suspend mode. To ensure delivery of events from non-wake-up sensors even when the screen is OFF, the application registering to the sensor must hold a partial wake-lock to keep the AP awAke, otherwise some events might be lost while the AP is asleep. Note that although events might be lost while the AP is asleep, the sensor will still consume power if it is not explicitly deactivated by the application.

2.2 Documents targeted at manufacturers

Sensors

- Suspend mode (non-wake-up and wake-up sensors)
- Batching (FIFO)

2.3 Sensor's type

Type by functionality.

Constant	sensor type	API level	others
TYPE_ACCELEROMETER	an accelerometer sensor type	3	
TYPE_ACCELEROMETER_UNCALIBRATED	an uncalibrated accelerometer sensor	O	
TYPE_ALL	all sensor types	3	
TYPE_AMBIENT_TEMPERATURE	an ambient temperature sensor type	14	
TYPE_GAME_ROTATION_VECTOR	an uncalibrated rotation vector sensor type	18	
TYPE_GEOMAGNETIC_ROTATION_VECTOR	a geo-magnetic rotation vector	19	
TYPE_GRAVITY	a gravity sensor type	9	
TYPE_GYROSCOPE	a gyroscope sensor type	3	
TYPE_GYROSCOPE_UNCALIBRATED	an uncalibrated gyroscope sensor type	18	
TYPE_HEART_BEAT	a motion detect sensor	24	
TYPE_HEART_RATE	a heart rate monitor	20	
TYPE_LIGHT	a light sensor type	3	
TYPE_LINEAR_ACCELERATION	a linear acceleration sensor type	9	
TYPE_LOW_LATENCY_OFFBODY_DETECT	a low latency off-body detect sensor	O	
TYPE_MAGNETIC_FIELD	a magnetic field sensor type	3	
TYPE_MAGNETIC_FIELD_UNCALIBRATED	an uncalibrated magnetic field sensor type	18	
TYPE_MOTION_DETECT	a motion detect sensor	24	
TYPE_ORIENTATION		3	Deprecated in API level 8
TYPE_POSE_6DOF	a pose sensor with 6 degrees of freedom	24	
TYPE_PRESSURE	a pressure sensor type	3	
TYPE_PROXIMITY	a proximity sensor type	3	wake up sensor
TYPE_RELATIVE_HUMIDITY	a relative humidity sensor type	14	
TYPE_ROTATION_VECTOR	a rotation vector sensor type	9	
TYPE_SIGNIFICANT_MOTION	a significant motion trigger sensor	18	wake up sensor
TYPE_STATIONARY_DETECT	a stationary detect sensor	24	
TYPE_STEP_COUNTER	a step counter sensor	19	
TYPE_STEP_DETECTOR	a step detector sensor	19	
TYPE_TEMPERATURE		3	Deprecated in API 14

Type by hardware feature

- **Non-wake-up sensors**

  Non-wake-up sensors are sensors that do not prevent the SoC from going into suspend mode and do not wake the SoC up to report data. In particular, the drivers are not allowed to hold wake-locks. It is the responsibility of applications to keep a partial wake lock should they wish to receive events from non-wake-up sensors while the screen is off. While the SoC is in suspend mode, the sensors must continue to function and generate events, which are put in a hardware FIFO. The events in the FIFO are delivered to the applications when the SoC wakes up. If the FIFO is too small to store all events, the older events are lost; the oldest data is dropped to accommodate the latest data. In the extreme case where the FIFO is nonexistent, all events generated while the SoC is in suspend mode are lost. One exception is the latest event from each on-change sensor: the last event must be saved outside of the FIFO so it cannot be lost.

  As soon as the SoC gets out of suspend mode, all events from the FIFO are reported and operations resume as normal.

  Applications using non-wake-up sensors should either hold a wake lock to ensure the system doesn't go to suspend, unregister from the sensors when they do not need them, or expect to lose events while the SoC is in suspend mode.

- **Wake-up sensors**

  In opposition to non-wake-up sensors, wake-up sensors ensure that their data is delivered independently of the state of the SoC. While the SoC is awake, the wake-up sensors behave like non-wake-up-sensors. When the SoC is asleep, wake-up sensors must wake up the SoC to deliver events. They must still let the SoC go into suspend mode, but must also wake it up when an event needs to be reported. That is, the sensor must wake the SoC up and deliver the events before the maximum reporting latency has elapsed or the hardware FIFO gets full.

  To ensure the applications have the time to receive the event before the SoC goes back to sleep, the driver must hold a "timeout wake lock" for 200 milliseconds each time an event is being reported. That is, the SoC should not be allowed to go back to sleep in the 200 milliseconds following a wake-up interrupt. This requirement will disappear in a future Android release, and we need this timeout wake lock until then.

2.4 How to get a wakeup or non-wakeup sensor

`SensorManager` provides three ways to get a `Sensor` (not dynamic)

```
Sensor getDefaultSensor(int type)
Sensor getDefaultSensor(int type, boolean wakeup)
List<Sensor> getSensorList(int type)
```

class Sensor public method get sensor class Sensor sensor
wakeup/non-wakeup sensor

```
Sensor getDefaultSensor(type,true/false)
```

wakeup/non-wakeup sensor null

```
List<Sensor> getSensorList(int type)
```

sensor wakeup non-wakeup sensor

```
Sensor.isWakeUpSensor(void)
```

sensor wakeup
Sensor-Types SensorManager calss SensorManager.getDefaultSensor(int type)

```
Sensor getDefaultSensor(int type)
```

wakeup/non-wakeup sensor null

- non-wakeup
  - SENSOR_TYPE_ACCELEROMETER
  - SENSOR_TYPE_AMBIENT_TEMPERATURE
  - SENSOR_TYPE_MAGNETIC_FIELD
  - SENSOR_TYPE_GYROSCOPE
  - SENSOR_TYPE_HEART_RATE
  - SENSOR_TYPE_LIGHT
  - SENSOR_TYPE_PRESSURE
  - SENSOR_TYPE_RELATIVE_HUMIDITY
  - SENSOR_TYPE_LINEAR_ACCELERATION
  - SENSOR_TYPE_STEP_DETECTOR
  - SENSOR_TYPE_STEP_COUNTER
  - SENSOR_TYPE_ROTATION_VECTOR
  - SENSOR_TYPE_GAME_ROTATION_VECTOR
  - SENSOR_TYPE_GRAVITY
  - SENSOR_TYPE_GEOMAGNETIC_ROTATION_VECTOR
  - SENSOR_TYPE_ORIENTATION (deprecated)
  - SENSOR_TYPE_GYROSCOPE_UNCALIBRATED
  - SENSOR_TYPE_MAGNETIC_FIELD_UNCALIBRATED
- wakeup
  - SENSOR_TYPE_PROXIMITY
  - SENSOR_TYPE_SIGNIFICANT_MOTION
  - SENSOR_TYPE_TILT_DETECTOR
  - SENSOR_TYPE_WAKE_GESTURE
  - SENSOR_TYPE_PICK_UP_GESTURE
  - SENSOR_TYPE_GLANCE_GESTURE

2.5 Sensor Registration and Unregistration

2.5.1 Registration

Class SensorManager sensor registration API

```
boolean registerListener(SensorEventListener listener, Sensor sensor,
                         int samplingPeriodUs)
boolean registerListener(SensorEventListener listener, Sensor sensor,
                         int samplingPeriodUs, int maxReportLatencyUs)
boolean registerListener(SensorEventListener listener, Sensor sensor,
                         int samplingPeriodUs, Handler handler)
boolean registerListener(SensorEventListener listener, Sensor sensor,
                         int samplingPeriodUs, int maxReportLatencyUs, Handler handler)
```

- listener: omitted
- sensor: omitted
- samplingPeriodUs:The rate sensor events are delivered at. Android System hint events deliver ( )
  - SENSOR_DELAY_NORMAL
  - SENSOR_DELAY_UI
  - SENSOR_DELAY_GAME
  - SENSOR_DELAY_FASTEST

  microseconds
- maxReportLatencyUs : sensor event event delivery microseconds 0 registerListener (listener, sensor, samplingPeriodUs)
- handler: the handler the sensor event will be delivered to
- : true if the sensor is supported and successfully enabled

2.5.2 Unregistration

Class SensorManager sensor registration API

```
void unregisterListener(SensorEventListener listener)
//unregisters a listener for all sensors
void unregisterListener(SensorEventListener listener, Sensor sensor)
//unregister a listener for the sensors with which it is registered
```

2.5.3 TODO

good practice TODO

- registerListener one shot trigger sensor such as SENSOR_TYPE_SIGNIFICANT_MOTION (sensor event continuous one shot ( ) TODO)
- samplingPeriodUs maxReportLatencyUs
  - non-positive
  - maxReportLatencyUs 0 FIFO buffer sensor events

2.6 Bizarre Usage Examples

API

2.6.1 On SensorManager Instance

```
m1=getSystemService(SENSOR_SERVICE) //get a SensorManager instance
m2=getSystemService(SENSOR_SERVICE) //get a SensorManager instance
//m1==m2
```

2.6.2 On Getting Sensors

```
m=getSystemService(SENSOR_SERVICE) //get a SensorManager instance
s1=m.getDefaultSensor(SENSOR_TYPE_ACCELEROMETER)  //non-wakeup
s2=m.getDefaultSensor(SENSOR_TYPE_ACCELEROMETER)  //non-wakeup
s3=m.getDefaultSensor(SENSOR_TYPE_ACCELEROMETER,false) //non-wakeup
s4=m.getDefaultSensor(SENSOR_TYPE_ACCELEROMETER,true)  //I have not got this
//s4==null && s1==s2 && s2==s3

s1=m.getDefaultSensor(SENSOR_TYPE_PROXIMITY)  //wakeup
s2=m.getDefaultSensor(SENSOR_TYPE_PROXIMITY)  //wakeup
s3=m.getDefaultSensor(SENSOR_TYPE_PROXIMITY,true) //wakeup
s4=m.getDefaultSensor(SENSOR_TYPE_PROXIMITY,false)  //non-wakeup
//s1==s2 && s2==s3 && s4!=null && s4!=s1

l1=m.getSensorList(SENSOR_TYPE_ALL)
l2=m.getSensorList(SENSOR_TYPE_ALL)
//l1==l2
```

2.6.3 On Registration

```
SensorEventListener l //suppose initialized
s=m.getDefaultSensor(SENSOR_TYPE_ACCELEROMETER)  //non-wakeup
m.registerListener(l,s,rate)
//debug: m.mSensorListeners.size = 1
m.registerListener(l,s,rate)
//debug: m.mSensorListeners.size = 1
m.unregisterListener(l,s1,rate) //function type of s1 != function type of s
//debug: m.mSensorListeners.size = 1
```

3 Processes and Threads

ref : link: dev > API Guides > App Components > Processes and Threads

By default, all components of the same application run in the same process and thread (called the "main" thread). However, you can arrange for different components in your application to run in separate processes, and you can create additional threads for any process.

3.1 Processes

In manifest file, each type of component element (e.g. `<activity>`) and `<application>` supports an `android:process` attribute that can specify a process in which that component should run.

3.2 Threads

A main thread is created when an application is launched. THsi thread is in charge of dispatching events to UI widgets and interactions with UI components from the Android UI toolkit. As such, the main thread is also called the UI thread.

There are two rules to Android's single thread model:

1. Do not block the UI thread. no events can be dispathed (including drawing) when blocked. blocking 5 seconds causes "application not responding",
2. Do not access the Android UI toolkit from outside the UI thread. they are not thread-safe.

3.2.1 Worker threads

Do blocking task in a new `Thread` and post UI update operations by methods such as `Activity.runOnUiThread(Runnable)` , `View.post(Runnable)`, `View.postDelayed(Runnable, long)`. Or use `AsyncTask` to make this process esaier.

3.2.2 TODO Thread-safe methods

3.2.3 Looper, Handler and Message

Looper runs a loop for a thread. It maintains a Message Queue, and get Message from the quese repeatly until the queue is empty.

An application has a main loop (source code link) that handles application lifecycle events etc..

3.3 TODO Interproceess Communication

Android offers a mechanism for interprocess communication (IPC) using remote procedure calls (RPCs), in which a method is called by an activity or other application component, but executed remotely (in another process), with any result returned back to the caller. Data must be understood by the operating system to be moved from the address space of one process to another.

3.4 Other Event based libaraies

ref : blog: Event Driven Android

- github: EventBus Android optimized event bus that simplifies communication between Activities, Fragments, Threads, Services, etc.
- github: Otto An enhanced event bus with emphasis on Android support

4 Lifecycle of components

ref : link: dev > API Guides > Process-lifecycle

"It is important that application developers understand how different application components (in particular Activity, Service, and BroadcastReceiver) impact the lifetime of the application's process. Not using these components correctly can result in the system killing the application's process while it is doing important work."

lifecycle bugs

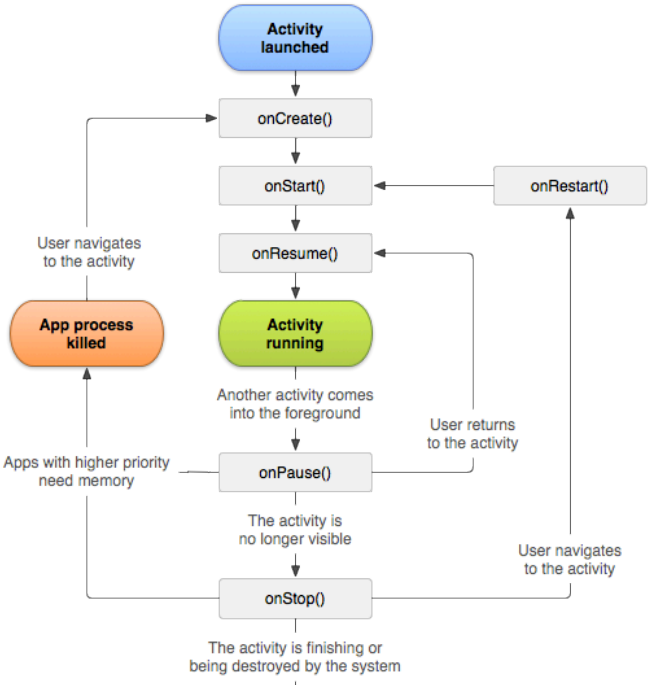
4.1 Activity

ref : link: dev > API Guides > App Components > Activities

For example, good implementation of the lifecycle callbacks can help ensure that your app avoids:

- Crashing if the user receives a phone call or switches to another app while using your app.
- Consuming valuable system resources when the user is not actively using it.
- Losing the user's progress if they leave your app and return to it at a later time.
- Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.

The *entire lifetime* is between `onCreate()` and `onDestroy()`. The *visible lifetime* is between `onStart()` and `onStop()`. The *foreground lifetime* is between `onResume()` and `onPause()`



Activity is a class. User extends this class will implement `onCreate()` to do their initial setup. You should always call up to your superclass when implementing these methods.

4.1.1 Starting Activities and Getting Results

- `startActvity(Intent)` : start an activity on the top of the activity stack, `Intent` describes the activity to be executed.
- `startActvitiyForResult(Intent, int)` : start an activity with a second integer parameter identifying the call.
- `onActivityResult(int, int, Intent)` : callback where you get the result.
- `setResult(int)` : use this to return data

4.1.2 Process Lifecycle Based on Activity Lifecycle

1. **Foreground activity** : visible, interactive
2. **Visible activity** : visible, not interactive, such as one sitting behind a foreground dialog.
3. **Backfround activity** : invisible, so the system may safely kill its process to reclaim memory.
4. **Empty process** : a process hosting no activites or other application components.

4.2 Service

Service is not a separate process, neither a thread. It is running in the application's main process unless otherwise specified. Service is a facility for the application to tell the system about something it wants to be doing in the background to expose some of its functionality to other apps.

4.3 Content Providers

Part II
Program Analysis

5 Related Work

About Sensor, GPS, WakeLock, or other energy related resources usage in Android Program.

5.1 What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps

link

Purdue University, static, MobiSys 2012, cited by 185

Wake lock, static, "the first to study energy bugs caused by wake lock leakage"(ELITE),

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 mis-handling 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 17 known instances of no-sleep bugs and found 34 new bugs in the 73 apps examined.

Bug Patterns:

- No-Sleep Code Path
  - forget to release the wakelock throught the code
  - some excution path doesn't have a release operation (**common**)
  - some higher level condition (like a deadlock) prevent the execution from reaching the release point
  - do not undertand the life-cycle of Android processes (**most common**)
- No-Sleep Race Condition, caused by race conditions in multi-threaded apps.
- No-Sleep Dilation, hold wkaelock for longer than needed.

Solution: dataflow analysis. by reaching definition
TO BE COMPLETED

5.2 Where has my battery gone? Finding sensor related energy black holes in smartphone applications

link

GreenDroid, dynamic, HKUST, NJU, PerCom 2013, Cited by 52

Yepang Liu

based on Java PathFinder(link) (a customizable virtual machine that enables the development of various verification algorithms)

assumption: sensory data should be efficiently used.

challenge: 1. Event driven programming paradigm; 2. how to analyze the utilization of sensory data.

" To address the first challenge, we derive an application execution model from Android specifications. This model captures application-generic temporal rules that specify calling relationships between event handlers. Enforcing these rules would enable JPF to realistically execute an Android application. To address the second challenge, we monitor an application's execution, and perform dynamic data flow tracking at a bytecode instruction level. "

Contributions:

- We propose a runtime analysis technique to automatically analyze sensory data utilization at different states of an Android application.
- We present an application execution model that captures application-generic temporal rules for event handler scheduling. This model is general enough to be used in other Android application analysis techniques.
- We implement a prototype tool called GreenDroid. To the best of our knowledge, GreenDroid is the first JPF extension that is able to verify Android applications.
- We evaluate GreenDroid using six popular Android applications. GreenDroid successfully located real energy inefficiency problems in four applications, and reported new problems for the remaining two.

5.3 Understanding and Detecting Wake Lock Misuses for Android Applications

link

ELITE, static, HKUST, NJU, Yepang Liu, FSE 2016

assumption: wake lock should protect critical computational task.

solution: static method

1. summarize the effect about wakelock and task for each top level method (callback for event).
2. Generate all possible sequences for top level method.

3. sequences should be valid.(components' lifecycle; some callback must be registered first).

4. Check.

Related Work worth reading
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.

5.4 Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications

link
HKUST, NJU, Yepang Liu, IEEE Software 2015
"Here, we discuss the challenges in diagnosing energy and performance bugs in real-life Android applications. We hope to inspire further efforts to adequately address these challenges. We also **review state-of-the-art diagnostic techniques and tools**. In particular, we offer results from our case study, which applied a representative tool to popular commercial Android applications and the Samsung Mobile Software Development Kit (SDK)."

5.4.1 State-of-the-art Diagnosis and Tools

Measurement and Estimation Techniques: measure or estimate app's energy consumption and performance.

- vLens(fine-graned source code level), eProf, PowerTutor
- Arvind Thiagarajan's framework for energy anlysis of rendering webpages
- Mantis's, precise performance models for app, estimate execution time.

Event Profilers

- ARO (Application Resource Optimizer) monitors cross-layer interactions—such as user events at the application layer and network packets at the system layer—to disclose inefficient radio resource usage
- AppInsight helps instrument smartphone applications' binaries to identify long latency execution paths
- Panappticon identies performance issues arising from inefficient platform code or problematic interactions among app.
- SunCat logs the events in a test run and summarizes event repetition patterns to help developers understand and predict performance problems

Pattern-Based Analyzers

- dynamic analyzers. ADEL (Automated Detector of Energy Leaks) and GreenDroid Execute an application and track program-data transformation, propagation and consumption.
- static.

- Abhinav Pathak's work;
- Lint, a popular static analyzer in the Android Studio SDK, detects a range of energy and performance bugs.
- PerfChecker can detect eight patterns of energy and performance bugs and provide actionable diagnostic information

End-User-Oriented Diagnosis(not for developers)

- eDoctor can correlate system and user events (such as conguration changes) to energy-heavy execution phases.
- Carat shares the same goal as eDoctor but adopts a collaborative, big-data-driven approach.

5.4.2 Disussion

- we should check for neccessity, not only energy cost.
- profilers can generate large profiles, visualization is desirable.(Maybe Machine Learning? gx)
- pattern based methods should find root causes.

5.5 Mining energy-greedy API usage patterns in Android apps: an empirical study

link

Univ of Molise, Italy, College of William and Mary, USA, MSR 2014, cited by 67

hardware power monitor.

Question: is sensor and GPS really energy-greedy? TO BE ANSWERED

Abstract:

Energy consumption of mobile applications is nowadays a hot topic, given the widespread use of mobile devices. The high demand for features and improved user experience, given the available powerful hardware, tend to increase the apps' energy consumption. However, excessive energy consumption in mobile apps could also be a consequence of energy greedy hardware, bad programming practices, or particular API usage patterns. We present the largest to date quantitative and qualitative empirical investigation into the categories of API calls and usage patterns that—in the context of the Android development framework—exhibit particularly high energy consumption profiles. By using a hardware power monitor, we measure energy consumption of method calls when executing typical usage scenarios in 55 mobile apps from different domains. Based on the collected data, we mine and analyze energy-greedy APIs and usage patterns. We zoom in and discuss the cases where either the anomalous energy consumption is unavoidable or where it is due to suboptimal usage or choice of APIs. Finally, we synthesize our findings into actionable knowledge and recipes for developers on how to reduce energy consumption while using certain categories of Android APIs and patterns.

5.6 Towards verifying android apps for the absence of no-sleep energy bugs

link and presentation and pdf

Univ of California, San Diego, short paper, USENIX 2012, cited by 31

Wakelock, pricise inter-procedural data flow analysis framework. *no-sleep* bugs. use WALA. no code.

Pollcy: (Component, Callback, Expected WakeLock state).

Data Fact: the wakelock instances at this program point it may hold. the set should be empty at certain program point(like **onPause** in Activity).

Asynchronous Calls: only explicit intents in which a single terget component is specified.

Lifecycles: the exit ICFG node of a callback method is connected to every entry CFG node of each of its successor method in the component's lifecycle.

Contributions:

- First, we have studied the lifecycle of different kinds of components in Android applications, and **use this information to precisely define a set of resource management policies specifying the correct usage of wake locks.**
- Second, compared to prior work we apply more precise analysis techniques for handling asynchronous calls, which are ubiquitous in Android.

- Third, to evaluate our techniques we run our tool on 328 real world Android applications and verify that 145 of those are exempt from the specific kind of no-sleep bug (w.r.t. our policies).

5.7 Static control-flow analysis of user-driven callbacks in Android applications

link

Ohio State University, Atanas Rountev, ICSE 15, cited by 56

5.8 WLCleaner: Reducing Energy Waste Caused by WakeLock Bugs at Runtime

WLCleaner, runtime fix. DASC 2014,

5.9 LEAKPOINT: Pinpointing the Causes of Memory Leaks

LEAKPOINT, conventional software, ICSE 2010

5.10 eDoctor Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones.

eDoctor, NSDI 2013

5.11 Effective Interprocedural Resource Leak Detection

Java, IBM, ICSE 2010

5.12 Learning Resource Management Specifications in Smartphones

ICPADS 2015

IMPORTANT, TO BE COMPLETED.

statically analyzes Android apps to mine resource management specifications (i.e., the correct order of resource operations)

Part III
Experiment

This part is about our experiment and system implementation.

6 Useful links

android-security-awesome

7 APK Collection

The dataset used in "ELITE"'s work.

8 APK analysis

8.1 Dalvik bytecode

official doc about dalvik bytecode link.

8.2 Apktool

link

A tool for reverse engineering Android apk files.
Freatures:

- Disassembling resources to nearly original form (including resources.arsc, classes.dex, 9.png. and XMLs)
- Rebuilding decoded resources back to binary APK/JAR
- Organizing and handling APKs that depend on framework resources
- Smali Debugging (Removed in 2.1.0 in favor of IdeaSmali)
- Helping with repetitive tasks

8.3 smali/baksmali

link

smali/baksmali is an assembler/disassembler for the dex format used by dalvik, Android's Java VM implementation. The syntax is loosely based on Jasmin's/dedexer's syntax, and supports the full functionality of the dex format (annotations, debug info, line info, etc.)