HYPNOS: Understanding and Treating Sleep Conflicts in Smartphones

Abhilash Jindal Abhinav Pathak Y. Charlie Hu Samuel Midkiff

Purdue University

{jindal0, pathaka, ychu, smidkiff}@purdue.edu

Abstract

To maximally conserve the critical resource of battery energy, smartphone OSes implement an aggressive system suspend policy that suspends the whole system after a brief period of user inactivity. This burdens developers with the responsibility of keeping the system on, or waking it up, to execute time-sensitive code. Developer mistakes in using the explicit power management unavoidably give rise to energy bugs, which cause significant, unexpected battery drain.

In this paper, we study a new class of energy bugs, called sleep conflicts, which can happen in smartphone device drivers. Sleep conflict happens when a component in a high power state is unable to transition back to the base power state because the system is suspended when the device driver code responsible for driving the transition is supposed to execute. We illustrate the root cause of sleep conflicts, develop a classification of the four types of sleep conflicts, and finally present a runtime system that performs sleep conflict avoidance, along with a simple yet effective pre-deployment testing scheme. We have implemented and evaluated our system on two Android smartphones. Our testing scheme detects several sleep conflicts in WiFi and vibrator drivers, and our runtime avoidance scheme effectively prevents sleep conflicts from draining the battery.

Categories and Subject Descriptors D.4.5 [Operating Systems]: Reliability.

General Terms Design, Experimentation, Reliability.

Keywords Smartphones, Mobile, Energy, Bugs.

1. Introduction

Only recently have computers had to contend with not having ready access to power. Servers and desktops have continuous access to power from the grid. Even battery powered laptops spend much of their time plugged in. The power management strategies for these devices reflect their ready access to power, and the processor and supporting devices, such as memory, remain on when in use and for some time thereafter. Mobile phones have emerged as the fastest growing computing platform, and for many people are their primary or only computing device. Unlike previous computers, their normal mode of operation is to be completely disconnected from an external power source, and their users expect them to run for at least a day before their batteries are exhausted. This usage model, combined with the relatively small amounts of energy available in mobile phone batteries, has led to radical changes in the power management model.

These changes in the power management model, and how they differ from the power model for traditional computing devices, can be summarized into two key points.

First, after a brief user inactivity, the phone enters the default state of system suspend, *i.e.,* the phone system-on-a-chip (SOC), including the CPU, ROM, and micro-controller circuits for various phone devices such as GPS, graphics, video and audio, is in a suspended state. The RAM is put into a self-refresh mode. In this state, the phone drains close-to-zero battery power. For example, on a Google Nexus One running Android 2.3, the CPU will attempt to initiate system suspend every 20ms, and will succeed unless one of the mechanisms discussed below is used by some device driver, an app, or the framework, to force it to abort the suspend process. Upon suspension, it consumes just 3mA of current. Second, the developers of apps and device drivers are expected to perform the power management of individual phone devices such as the GPS, WiFI, 3G, vibrator, and so forth, and to ensure that when a change in the power state of a device is needed the SOC is on, and hence the CPU can be woken up even if it is in a low-power sleep state. Three mechanisms can be used to keep the system from being suspended: *wakelocks*, which forbid system suspend whenever one is held; suspend notifiers, which are callback functions registered by individual devices to be consulted for permission to suspend the SOC at the last moment of each suspend process; and hardware wakeups, such as the Real

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

Time Clock (RTC), which when fired wake up the SOC and the CPU.

This explicit power management philosophy, employed in modern smartphones and induced by the default "off" policy, leads to a significant burden on programmers. Programmers are forced, for the first time, to think about power management as a first-order concern for the correct behavior of apps. The failure to deal with the added complexity of the new power management paradigm is manifested in the CPU and various phone devices such as the GPS, vibrator and LED being on when they should be off, rapidly draining batteries. Collectively such mistakes in explicit power management by the developers which cause significant, unexpected energy drain are known as energy bugs [20].

In this paper, we study a new class of energy bugs, called *sleep conflicts*, that happen in smartphone device drivers. Sleep conflicts arise when a hardware device is transitioned to an active power state by a software entity (*e.g.,* device driver) and the system is then suspended. As a result, the software code that is responsible for bringing the device back to a low power state (*e.g.,* sleep state) cannot run since the CPU is suspended, and consequently the device stays in the high power state indefinitely. Sleep conflicts can result in rapid draining of the battery and significant user frustration.

This paper directly attacks the problem of pre-deployment testing for sleep conflicts and runtime discovery and amelioration of the effects of existing sleep conflict bugs. A deep understanding of the root causes of sleep conflicts, acquired through the study of tens of thousands of lines of device driver code and their interplay with the power management system, and a classification of power transition scenarios in device drivers where sleep conflict bugs can manifest themselves, have allowed us to develop two novel runtime techniques. The first implements a simple, yet elegant, black-box runtime technique that enables easy testing of drivers before they are released and the second detects, at runtime, pending sleep conflicts and avoids them.

The contributions of the paper are as follows.

- First, we present a classification of all the scenarios where sleep conflicts can arise in device drivers, by analyzing power state transition rules of the FSM power models of smartphone devices implemented by the drivers. Such a classification provides insights into the root cause of sleep conflicts, and gives hints to detecting and mitigating sleep conflicts at runtime.
- Second, we present a light-weight sleep conflict avoidance runtime system that accurately detects pending sleep conflicts during device driver execution and intervenes to avoid them from being triggered.
- Third, we present a novel technique that builds on top of our pending sleep conflict detection scheme for comprehensively testing sleep conflicts in device drivers. Our testing scheme manipulates the default user inactivity timeout values to expose false negatives in runtime sleep conflict detection, *i.e.,* potential sleep conflicts that are shielded behind the inactivity timeout.
- Fourth, we have implemented the runtime avoidance and testing system, HYPNOS, in Android and used it to test device drivers of two smartphones. Testing using HYPNOS identified 5 instances of sleep conflicts in the vibrator, compass, touchscreen and WiFi drivers.

2. Power Management in Smartphones

We start with an overview of power management basics in modern smartphones.

2.1 Phone Devices have Multiple Power States

A modern smartphone platform consists of the SOC which includes the CPU, RAM, ROM, and micro-controller circuits for various phone devices such as GPS, graphics, video and audio, and a wide variety of hardware I/O *devices* for enriching the user experience. Typical devices include the CPU, LCD, memory, SD Card (sdcard), networking devices such as the WiFi NIC, cellular and bluetooth, and a wide variety of sensors such as the GPS, one or more cameras, accelerometer, digital compass, touch sensors, microphone, and speakers. Unlike desktop and server machines, on smartphones many of these devices consume a comparable amount of power as the CPU, and thereby contribute to a significant portion of the total power consumption.

The default power state of SOC is suspend, where all components on the SOC, including the CPU, ROM, and micro-controller circuits for various phone devices such as GPS, graphics, video and audio, are suspended, RAM is put in a self-refresh mode, and the SOC drains close-to-zero battery power. When in active use, SOC can be in the active power state or one or more standby states where different components (*e.g.,* the CPU) can enter lower power states, but various timer and interrupt events from the devices will move the needed SOC components back to the active state.

Independently from the SOC state, a hardware device can be in several operating modes, known as different *power states*, with each consuming a different amount of power. The motivation for having multiple power states is to achieve power-proportionality, that is, the power consumed by a device should be commensurate with its active utilization. Smartphones support power proportionality in three ways. First, smartphone CPUs support frequency scaling. For example, the ARM processor on Google Nexus One has seven different frequencies while in active power state. Further, the Saltwell CPUs on Intel's Medfield phones can enter one of several levels of sleep states (C1–C6) [23]. Second, each of the phone devices is in an active power state (D0) during active use.[1] After active use, it may linger in one or more ac-

[1] D0-D3 is a standard platform independent power management terminology (ACPI).

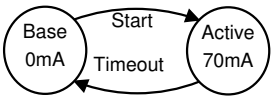


Figure 1: Power state transitions for the vibrator.

tive idle states (*e.g.,* D1, D2), known as tail states [9, 14, 19]. This has been observed in several phone devices, including GPS, WiFi, sdcard, 3G, and 4G. Finally, during a system suspend, the device should be put into a suspend state (D3) by the device driver, and consume practically zero power.

2.2 Device Drivers Implement Power State Transitions

Power management in Linux (Android kernel) is split between OS power management and Linux runtime power management. The OS power manager directs the SOC to various power states (*e.g.,* system suspend) depending on the policy. Additionally, Linux kernel 2.6.33 and beyond require all device drivers to implement Linux runtime power management, through which device drivers can implement autonomous power management when devices are idle.

Smartphone device drivers implement various "smarts" in manipulating the transitions among different power states of their devices. In principle, all the power states and transition rules of a phone device can be uncovered from reading its driver. However, most of the device drivers for smartphones, including Android handsets, are proprietary. Pathak *et al.* [19] presented a system-call based finite state machine (FSM) power model that accurately captures both utilization-based and non-utilization-based power behavior of smartphones, and a systematic methodology for deriving the power model for all the devices on a given phone.

Figure 1 shows the state machine involved in driving the power behavior of the smartphone vibrator. The vibrator has just two power states: on and off. Its device driver transitions the device from the off state when the app explicitly invokes the vibrator API, and transitions the device back to the off state after the timeout period specified in the API call.

An Example Driver. Listing 1 shows the vibrator driver used on the Motorola Spica phone. The API exported to the user-space app simply writes the number of milliseconds for which the phone should vibrate in file `/sys/class/timed_output/vibrator/enable`. As soon as a value is written into this file, the `enable_vibrator_from_user` callback function registered by the driver (line 4) is passed the value and starts to run. This function starts the vibrator (line 18) and registers a callback function with a high-resolution timer (line 20). The high-resolution timer generates an interrupt as soon as the timer expires, and the kernel schedules the driver defined timer callback function `vibrator_timer_func` to run (line 10). High-resolution timers are supported by newer timer implementation that offers higher accuracy (on the order of microseconds) than conventional system timer ticks (on the order of milliseconds). Conventionally, when a process registers a timer event, like sleeping, the time at which the event will be triggered is rounded off to the nearest system timer tick interval leading to inaccuracies on the order of milliseconds, which is unacceptable for real-time user applications or device drivers. High resolution timers, on the other hand, enable events to be as accurate as the hardware allows (typically 1 microsecond). However, unlike the Real Time Clock, they can not wake up the SOC from the suspended state.

Listing 1: Vibrator driver that contains sleep conflict.

```
1  static struct timed_output_dev timed_output_vt = {
2    .name     = "vibrator",
3    .get_time = get_time_for_vibrator,
4    .enable   = enable_vibrator_from_user,
5  };
6  // Called when the device is initalized
7  static void __init vibrator_init(void) {
8    // Register the timer
9    hrtimer_init(&timer, CLOCK_MONOTONIC,
         HRTIMER_MODE_REL);
10   timer.function = vibrator_timer_func;
11   // Register the device
12   ret = timed_output_dev_register(&timed_output_vt
         );
13 }
14 // Called as soon as user writes a value into
       enable file
15 static void enable_vibrator_from_user(struct
       timed_output_dev *dev,int value) {
16   if (value > 0) {
17     // Start vibration
18     vibrator_start();
19     // Register a timer to fire after value
           milliseconds
20     hrtimer_start(&timer,ktime_set(value/1000,(
           value1000*1000000),HRTIMER_MODE_REL);
21   }
22 }
23 // Automaticallly called when the registered timer
       expires
24 static enum hrtimer_restart vibrator_timer_func(
       struct hrtimer *timer) {
25   // End the vibration
26   vibrator_end();
27   return HRTIMER_NORESTART;
28 }
```

3. Rise of Sleep Conflicts

In this section, we explain the philosophy behind the smartphone sleep policies that give rise to sleep conflicts.

3.1 Aggressive Sleeping Policy

The push for maximal energy savings on smartphones quickly drove their OSes, such as Android, to pursue an aggressive system *sleeping policy*.[2] After a brief period of user inactivity, the OS power management lets the phone SOC enter its default state, *system suspend*. The intuition behind the aggressive sleeping policy is that smartphone usage is predominantly user-interactive, and hence a short user inactivity suggests a potentially long period of user inactivity, during which presumably all phone devices should be put to sleep.

[2] In this paper, we use system suspend and system sleeping interchangeably.

The aggressive sleeping policy is implemented by tracking user interactions with the phone. More precisely, *user inactivity* is defined as a time period during which there is a lack of input by the user, *i.e.,* the user has not touched the screen or any peripheral buttons on the device. Two user-configurable timers are used to control the screen and the CPU (full system suspend) sleeping policy. When the first timer fires, the screen is dimmed, and when the second timer is fired, the screen is shut down and the CPU initiates the system suspend process. Clearly, the timeout values affect the balance between battery drain and user inconvenience.

After entering system suspend, the SOC can be woken up by hardware wakeups such as the Real Time Clock (RTC), power button push by the user, WiFi wake-on-lan, or an incoming phone call. Background services, such as syncing, health monitoring *etc.*, need to run periodically, and rely on the RTC to wake up the SOC and CPU. However, during transient wakeups caused by background services, the user-inactivity timers are not used, and the SOC will quickly suspend again unless the awakened service explicitly invokes some mechanisms to keep the system on (Section 3.4).

3.2 Time-Critical Sections

By their very nature, the code segments in device drivers that perform power state manipulations are *time-critical* sections; if the CPU is suspended at the moment that such a section should run, it will *not* be able to run and hence cannot perform the intended power transition of the device. For example, the callback function in the vibrator driver in Listing 1 (line 24) is a time-critical section as it controls the transition from the active power state back to the suspend power state (line 26), ending the vibration.

3.3 Sleep Conflict

The three factors, (1) a device can be in one of several power states independently from the SOC power state; (2) the device driver controls power state transitions; and (3) the OS employs an aggressive system suspend policy; together give rise to a class of new energy bugs, which we call *sleep conflicts*. Sleep conflict happens when a device in a high power state is unable to transition back to the base power state because the CPU is asleep and the device driver cannot make progress in its execution to drive the transition. These bugs can drastically reduce the standby times of the phone. As described in Section 3.1, the aggressive sleeping policy helps the phone to achieve long standby times because the phone consumes near zero power in the suspended state (3mA on Google Nexus One phone) and stays in this suspended state for most of the time. But due to a sleep conflict bug, the standby power consumption can be increased by 10 to 25 times. For example, the bug we found in the vibrator of Google Nexus One causes it to drain 40 mA in the standby mode.

We now illustrate a sleep conflict bug using a simple phone vibrating app that triggers the sleep conflict in the vibrator. This app is partially shown in the code snippet in Listing 2. The code simply starts a new thread which intends to sleep for 20 seconds and then vibrate the phone for 10 seconds. The phone is configured in such a way that it performs a system suspend after 15 seconds of user inactivity (the default on Android). Once the app is started, the following sequence of events take place, as shown in Figure 2: (a) The app sleeps for the intended period of 20 seconds. (b) At 15 seconds into the sleep, the system sleeping policy triggers a system suspend because of user inactivity, and the system may easily sleep beyond 20 seconds from the start of the app. (c) Recall that the SOC/CPU may wake up occasionally for short intervals due to miscellaneous background services that registered RTC timers. When the SCO/CPU first wakes up after the 20 second sleep interval, the app wakes up and executes the next statement (line 3) which starts vibrating the phone for an intended period of 10 seconds, by writing 10,000 milliseconds in file `/sys/class/timed_output/vibrator/enable` (Section 2.2). Recall this action causes the `enable_vibrator_from_user` callback function registered by the driver in Listing 1 to start to run. (d) When the SOC/CPU wakes up for a background activity, it will go back to sleep in 20ms unless a wakelock is held to keep it awake. We note even if the brief background activity acquires a wakelock, other tasks cannot make any assumptions about how long that wakelock will be held and the SOC/CPU will stay awake. Soon (20ms) after no wakelock is held, the SOC/CPU will go into the suspend state, while the phone is still vibrating. (e) As a result, at the end of the intended vibrating duration of 10 seconds, the SOC/CPU may be in the suspend state, and the device driver cannot run to stop the vibration. This instance of sleep conflict arises because the high resolution timer used in `enable_vibrator_from_user` in the vibrator driver (Listing 1), unlike the RTC, cannot wake up the CPU from the suspend state, and hence cannot trigger the callback function that ends the vibration if the CPU is suspended at the end of the timer duration. (f) The phone continues vibrating till the next time the SOC/CPU wakes up, which could happen long after the intended 10 seconds. In our experiments running this app, we have observed that the phone vibrates from exactly 10 seconds up to 15 minutes!

Listing 2: Sleep conflict example: Code showing how the vibrator set for 10 seconds can potentially vibrate for several minutes.

```
1 try {
2     Thread.sleep(20000); //Sleep for 20s
3     vib.vibrate(10000);  //Vibrate for 10s
4 } catch (InterruptedException e) {
5     e.printStackTrace();
6 }
```

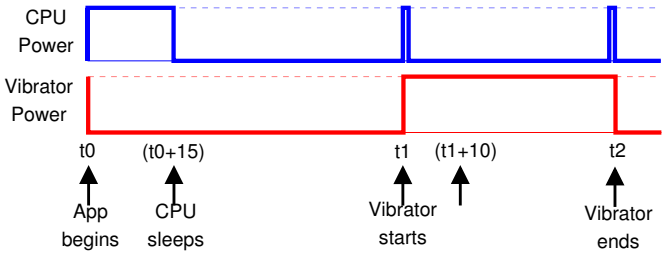


Figure 2: The power consumption of the CPU and vibrator after the app is started at time t0.

3.4 Mechanisms for Keeping the System Awake

To support background services as well as non-interactive foreground jobs under the aggressive CPU sleeping policy, the Linux kernel power manager on Android, along with hardware support, provides 3 general mechanisms to keep the system from suspending or to wake it up. In principle, the device drivers can use these to prevent the system (and the CPU) from being suspended when the drivers need to transition the devices back to the suspend states, and hence *prevent* sleep conflicts. However, these mechanisms are low-level and hard to use, and developer mistakes in using them in drivers give rise to sleep conflicts. For example, we studied four vibrator drivers that come with four phones, Motorola Spica, Motorola Captivate, HTC Nexus One and H1droid, respectively. Only the H1droid driver properly uses one of the mechanisms (suspend notifiers) to avoid sleep conflict.

Wakelocks. Wakelocks provide a high-level mechanism for preventing SOC suspend. They are special objects with two associated APIs calls, acquire and release, with the following semantics. If any process acquires a wakelock, the SOC cannot suspend. Further, a wakelock acquire can take a timeout parameter, signifying the wakelock is automatically released after the specified timeout period. Wakelocks can be used by both kernel-space and user-space programs. Finally, a wakelock object needs to be initialized, *e.g.,* by calling the kernel-space API `wake_lock_init()`, which registers the wakelock with the kernel wakelock manager.

Suspend Notifiers. Suspend notifiers provide a low-level mechanism for various entities in the kernel, device drivers in particular, to abort or allow each SOC suspend. A suspend notifier is a callback function that a device driver registers with the kernel power manager, which maintains a list of pointers to all the notifiers. When the kernel power manager initiates an SOC suspend process, the very first thing it checks is if any wakelock is not released, which would abort the process. Otherwise, the list of all suspend notifiers are called one by one. If any notifier returns an error, the suspend process is aborted. If none of the suspend notifiers returns an error, the SOC is suspended.[3]

[3] We note that CPU wakelocks are implemented at the low level by registering a notifier with the kernel that always denies the suspend request.

Listing 3: Ensuring the CPU is on during time critical sections using a suspend notifier.

```
1  // Global variable keeping track of current state
       of vibrator
2  int isVibrating = 0;
3  static void vibrator_start(void){
4    isVibrating = 1;
5    ..
6  }
7  static void vibrator_end(void){
8    ..
9    isVibrating = 0;
10 }
11 // Suspend notifier called automatically at CPU
       suspend
12 static int vibrator_suspend(struct platform_device
       *pdev, pm_message_t state){
13   if(isVibrating)
14     vibrator_end();
15 }
16 struct platform_driver vibrator_driver = {
17   .suspend    = &vibrator_suspend,
18   ..
19 };
20 // Called when the device is initalized
21 static void __init vibrator_init(void) {
22   platform_driver_register(&vibrator_driver);
23   ..
24 }
```

Hardware Wakeup. While wakelocks and suspend notifiers prevent system suspend, hardware wakeups provide the mechanisms for waking up the SOC from the suspend state. A general hardware wakeup mechanism is the Real Time Clock (RTC). The mechanism is essential to support services that should run at fixed time intervals. Any kernel process can set an alarm with the RTC using `rtc_set_alarm(struct device*, struct rtc_time*)` and read the next scheduled alarm time using `rtc_read_alarm(struct device*, struct rtc_time*)` [4]. Internally, the RTC maintains all the registered alarm times. The alarm initialization API, `alarm_init()`, specifies a callback function which will be executed after the timer fires. The RTC API is also exported to the user space, which enables apps, like an alarm clock, to perform an action such as playing alarm at a user-specified time.

Additionally, several smartphone devices can directly wake up the SOC from the suspend state, including the power button when pushed by the user, WiFi wake-on-lan, or incoming phone calls. The wake-on-lan feature is essential for the perceived persistence of long-standing sockets on the phone, and wakeup by incoming phone calls enables the perception that the phone is always on.

Example Use of the Mechanisms. Listing 3 shows how to use suspend notifiers to avoid sleep conflict. A flag `isVibrating` is maintained at all times and keeps track of the current state of the vibrator. In `vibrator_start` the flag is set to 1, and in `vibrator_end` it is reset to 0. A

[4] Android (and later, following Android's lead, POSIX) provides a simple hardware-wakeup alarm timer API that makes use of suspend notifiers, high-resolution timers and RTC wakeup.

suspend notifier, `vibrator_suspend`, is also registered with the kernel. When the CPU is about to suspend, the notifier is called. If the flag is on, the vibrator is vibrating, and the notifier can (1) stop the vibrator and allow CPU to go to sleep (shown in Listing 3); (2) abort the suspension process by returning an error; or (3) insert an RTC timer to wake up the system when the vibrator is expected to be turned off. We note that any of the options prevents sleep conflict, and which to use is a policy decision that can potentially vary among device drivers.

4. Classification of Sleep Conflict

To gain insight into the root cause of sleep conflict bugs in device drivers, we systematically analyze the power state transitions of the FSM power models of smartphone devices, and derive all the possible scenarios where sleep conflicts can arise. Such a classification provides insights into the root cause of sleep conflicts and gives hints on how to detect and mitigate impending sleep conflicts at runtime.

The first column of Table 1 shows the four possible scenarios where sleep conflicts can arise in the drivers for modern smartphone devices. We discuss each of these in turn.

Scenario 1: Waiting for end of active utilization. A system call from an app performing I/O operations, *e.g.,* writing to 3G or sdcard, can activate the device to enter an active power state for a long enough period of time to carry out the work specified in the system call, *e.g.,* the more data that needs to be transmitted, the longer the period the device is activated. For example, the power model for 3G on the Nexus One in Figure 3 shows the 3G device transitions among a base state, an active state, and a tail state. If the system call is triggered by a background job that woke up the SOC/CPU, and there is no user interaction with the device, and hence no user-inactivity timers are used, the SOC may suspend within 20ms. In that case, the device driver code responsible for transitioning the device into a lower power state, *e.g.,* the tail state in the case of 3G, will not be able to run. If this happens, the device will stay in the high power state till the next time the SOC wakes up, unnecessarily using energy and draining the battery.

Scenario 2: Waiting for a timeout transition. After a period of active utilization, there are three possible ways a device is transitioned to the suspend power state. The simplest form is *timeout*. As discussed earlier, several phone devices, including the GPS, WiFi, sdcard, 3G, and 4G, have idle power states known as tail states [9, 14, 19]. After active utilization, these devices transition into, and linger in, tail states for a fixed period of time. For example, the power model for 3G in Figure 3 shows that after active use, the 3G radio enters and stays in the tail state for 6 seconds. Since the driver may not have any work to do until the end of this period, it typically registers for a wakeup timer with the kernel and goes to sleep. The intention is that when the timer expires, the device driver wakes up and executes the code that puts the device into the suspend power state.

If, however, the CPU is asleep when the device is to be suspended, *e.g.,* the wakeup timer was a software timer or a high-resolution timer, neither of which can wake up the system from suspend, the device driver code cannot run and hence cannot put the device back into the suspend state. When this happens, the device can stay in the tail power state indefinitely, leading to an instance of a sleep conflict.

Scenario 3: Waiting for a program-input transition. A third way of transitioning into the suspend power state from active utilization is explicit *program input*. An app can tell the driver to end active utilization. For example, with LED and flashlight devices, an app directly sets the brightness of the LED by writing into `/sys/class/leds/<led>/brightness`. When the brightness is set to zero by the user program, the driver, implementing the power management state machine for LEDs and flashlight (Figure 4), turns off the LED. Similarly, with 3G and WiFi, if the user program closes the last socket, the drivers send their respective devices into the suspend power state.

If, however, the SOC/CPU are suspended before executing the explicit API invoked by the user-space program, the API will not be executed and hence cannot give the appropriate command(s) to the device driver to turn off the device. If the only transition out of this state is triggered by program input, the device would stay in high power and continue to drain energy.

Scenario 4: Waiting for an external event transition. The last trigger to transition a device from a high power state to the suspend state is an *external event*. To improve the energy efficiency without giving up on performance and functionality, some modern day devices come with a few built in ”smarts”. For example, the WiFi device can stay connected with the AP while responding to beacons, without any help from the CPU, letting it to suspend in the meantime. Additionally before the CPU suspends, the WiFi driver can load an appropriate packet filter into the WiFi device and set it up as a wakeup source. Now, if the WiFi NIC receives a new packet from the network, it can wake up the CPU (wake-on-LAN) to let it process the new packet. Similarly, many drivers, such as SDcard, make use of the direct memory accesses (DMA) support in the hardware to save energy. During large data transfers, the driver just specifies the appropriate memory location and the amount of data to copy to the device and configures it to be a wakeup source. The device, after finishing the data transfer, wakes up the CPU which will send more data transfer requests or shut down the device.

But, the sleep conflict arises if a device driver fails to configure its device as a wakeup source. In this scenario, even if the event, external to the CPU (*e.g.,* packet reception, data transfer completion), occurs, the device cannot wake

Table 1: Power state transition scenarios that can lead to sleep conflicts, and generic mechanisms for preventing system suspend that can be used in device drivers.

Power state transition scenario	Example device driver	Suitable mechanism for keeping system awake
(During active utilization)		
Case 1: waiting for end of utilization (EoU)	3G, WiFi, sdcard	Wakelocks, Suspend notifiers
(After active utilization)		
Case 2: waiting for timeout transition	WiFi, Vibrator	Hardware wakeup
Case 3: waiting for program-input transition	WiFi, 3G, LED, GPS	Wakelocks, Suspend notifiers
Case 4: waiting for external-event transition	WiFi wake-on-lan	Hardware wakeup

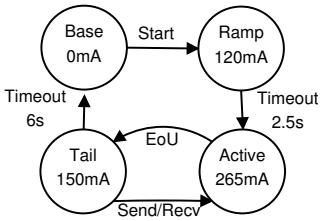


Figure 3: Power state transitions for 3G on Nexus One.

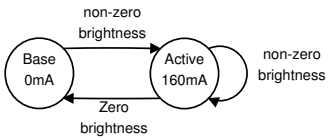


Figure 4: Power state transitions for LED/flashlight. Current shown is for flashlight on Nexus One.

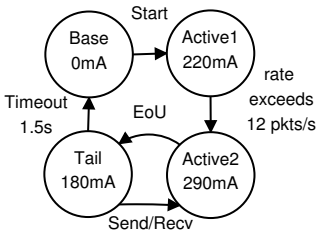


Figure 5: Power state transitions for WiFi on HTC Magic.

the CPU up, and will stay stuck in the high power state unnecessarily.

5. HYPNOS

In this section, we present the design and implementation of HYPNOS, a runtime system that performs sleep conflict bug avoidance in smartphone device drivers.

5.1 Design Space

Sleep conflict resembles the deadlock problem in resource allocation in operating systems. In deadlock problems, multiple entities hold some resources while requesting more, and hence none can make progress. In sleep conflict problems, the CPU and a device are racing to sleep at the moment the device is supposed to be put to sleep. If the CPU sleeps first, it is analogous to holding the resource (*i.e.,* the CPU) that the device needs (to run the time-critical driver code) to sleep, leading to a "deadlock" scenario.

One approach to finding and correcting sleep conflict bugs is static compiler analysis of each device driver's code. Such an approach faces two challenges. First, most device drivers are third-party software and their proprietary code is not available to a static tool. Second, even if the device driver source code is available, statically analyzing them is extremely difficult, for the following reasons. (1) Many device drivers are in a parent-child relationship. As a result, a device driver can hold wakelocks for one or more other device drivers. Deducing these relationships is tricky, especially when drivers are accessed via function pointers in an OS structure or code is not available for all drivers. (2) In the low level driver code, there are many function pointers that are passed between functions. Statically deducing the exact function referred to by these pointers, and hence the control flow of the program, is hard. (3) Some buses send messages to perform power state transition of the devices, *e.g.,* USB. Statically deciphering the semantics of these messages to resolve what power state transition is being requested will be difficult. All three reasons will lead to a conservative analysis and false positives.

As with deadlocks, another approach to treating sleep conflicts is avoidance. This is done by checking conditions that can lead to a sleep conflict at every pending system suspension. A major advantage of this approach is that it does not require device driver source code, as long as it can monitor power transitions and have access to the FSM power model for each device.

Another potential design choice is testing via symbolic execution, which recently has been applied to testing device drivers for reliability bugs (*e.g.,* [12, 15, 24]). Symbolic execution executes the driver code on all possible device inputs, providing thorough coverage of driver code.

We design HYPNOS to perform both online sleep conflict avoidance and pre-deployment testing. However, instead of applying symbolic execution, we explore the timing nature of sleep conflict bugs and the timing of the system suspend mechanism on smartphones to devise a simple, light-weight, yet effective, testing scheme for sleep conflicts.

5.2 Key Observation

The design of HYPNOS exploits a key observation that even though the device drivers are closed-source, both their interfaces with user programs above, and bus drivers below, are

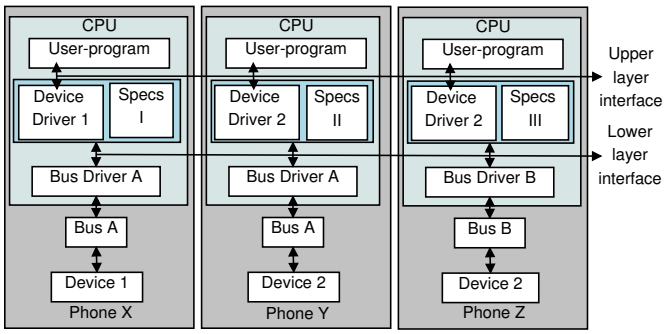


Figure 6: Phones X,Y and Z running the same user program.

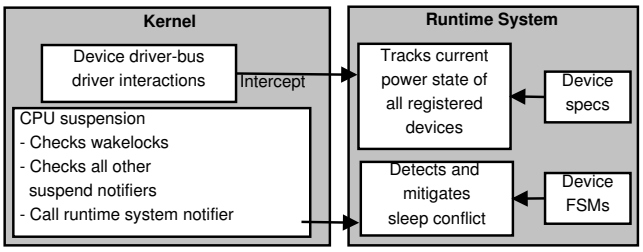


Figure 7: The HYPNOS architecture. The system monitors power state transitions by intercepting bus driver interactions, checks for sleep conflict conditions right before system suspend, and performs avoidance actions accordingly.

fixed and can be determined by examining the Linux kernel code. The fixed interfaces allow our runtime system to log device drivers' interactions with entities above and below to (1) deduce the current power state of the device and (2) check conditions for potential sleep conflicts.

The reasons the interfaces are generally fixed in OSes is to support portability of programs across different drivers for similar devices and to support portability of device drivers across different *buses* (*e.g.,* PCI and USB) used by drivers to communicate with devices. Figure 6 shows a user program running on three different phones X, Y and Z accessing a device instantiated with different hardware. The fixed *upper layer interface* encapsulates the different drivers in a standard interface, and in Android this interface is presented by device drivers to user programs via *device classes*. Device classes provide uniform handling of functionally for similar devices, *e.g.,* an accelerometer, a touch controller, and a joystick all fall under the `input` device class.

Devices communicate with the CPU via buses, *e.g.,* PCI and USB. Since the same device driver may have to communicate with the different bus chips on different phones, fixed *lower layer interactions* with bus drivers are used to encapsulates the different bus drivers in a standard interface. The device driver loads different specifications (specs) for the different bus chips; each spec maps the parameters passed by the device drivers to the actual device accessed via the bus driver. As a synthetic example, in Figure 6 phones Y and Z have the same device 2 but different bus driver chips A and B. Device driver 2 loads specs II when on phone Y and loads specs III when on phone Z. As a real example, in Android, (lower layer) API `gpio_set_value(unsigned gpio, int value)` exported by the GPIO bus driver is used by device drivers to set an output pin to high or low, depending on the value passed. This same interface will be used by all the device drivers maintaining devices on GPIO, such as the vibrator, and LEDs, which are distinguished by the different `gpio` values specified when calling the API. Similarly, `pci_set_power_state(struct pci_dev *dev, pci_power_t state)` is an API exported by the PCI bus driver to device drivers above to set the power state of power management-enabled PCI devices.

5.3 Design of HYPNOS

HYPNOS provides support for both debugging device drivers before they are deployed, and for catching sleep conflicts in smartphones as they happen, to avoid the power loss resulting from a sleep conflict. Recall that a sleep conflict occurs when the SOC/CPU is in the suspend state when a driver needs to transition its device back to the suspend state. To both detect sleep conflicts before deployment and to avoid sleep conflicts in deployed systems, the HYPNOS runtime must check to see if the necessary conditions for a sleep conflict can exist. That is, at some time when the SOC/CPU is about to sleep, check (1) if all devices are already in the suspend state, and (2) if some are not, determine if their drivers have invoked some mechanism to ensure that the system will wake up at the intended time for the power state transition. HYPNOS takes as input the bus driver specs (needed for logging power related interactions between device drivers and bus drivers) and the FSM power models for all the phone devices. The FSM models for the devices of a given phone are reverse-engineered offline [19].

First, to enable checking of condition (1) (if a device is in an active state), the runtime system maintains a data structure containing the current power state of all the devices, along with their FSM power models. The current power states are gathered by intercepting the power related interactions between the device drivers and the bus driver via the lower layer interface (Section 5.2).

Second, to enable checking of condition (2) (if a device is using any of the mechanisms described above to ensure the CPU will be awake when a power transition is needed) the following actions are taken. At system boot-up, the HYPNOS runtime system registers a suspend notifier for itself with the kernel power manager. In particular, it makes sure that this notifier is registered after the suspend notifiers of the wakelock implementation and after all of the devices have been registered. This trick ensures that HYPNOS's suspend notifier will run only after the notifiers of all the devices and the wakelock implementation have already run and given the "green light", and hence the system is really about to suspend.

Algorithm 1 Pseudo-code of the HYPNOS runtime system

```
for all devices not in base state do
  if next transition is end of utilization then
    return false
  end if
  if next transition is timeout then
    next wakeup time = rtc get alarm(..);
    if next wakeup time > timeout time then
      rtc set alarm( timeout time );
    end if
  end if
  if next transition is external event then
    if device has not registered hardware wakeup then
      return false
    end if
  end if
  if next transition is user input then
    return false
  end if
end for
return true;
```

Online Avoidance. HYPNOS performs online sleep conflict avoidance in deployed systems. When its suspend notifier is invoked at the end of the system suspend process, it checks for the two conditions necessary for sleep conflict and performs avoidance as shown in Algorithm 1. For each device that is currently not in the suspend state, the notifier checks the device's FSM power model. (1) If the transition out of the current state depends on an external condition, the system checks whether the driver has registered a hardware wakeup. For example, it checks if the driver has invoked the corresponding API such as `enable_irq_wake()` for GPIO signals; the checking is by polling `device_may_wakeup()` exported by the kernel power manager. If this has not happened, suspending the CPU will result in a sleep conflict. To avoid this, HYPNOS aborts the system suspend process by returning an error from its notifier. (2) Next, if the transition out of the current state depends on a timeout, the system checks when the next RTC hardware wakeup (if any) is scheduled, via the `rtc_read_alarm()` API. If it is much later than the timeout specified in the FSM, there is a potential sleep conflict. To avoid the sleep conflict, the system itself schedules an RTC hardware wakeup for the remaining time until the timeout specified in the FSM. (3) If the transition from the current state depends on a trigger from some user-space program, there is a potential sleep conflict because the user-space program cannot run after system suspend. The notifier again aborts the suspend process by returning an error.

Whenever the runtime system performs a sleep conflict avoidance action, the current power state and pending transition of the device causing sleep conflict are logged and reported to the driver developer at a later point when the phone is idle and has good connectivity.

Pre-Deployment Testing. Testing device drivers for sleep conflict, especially when their code is not available, is in general hard because of the large number of usage patterns of the devices arising from different sequences of events with different timing. In the following, we exploit a key observation of the system suspend process to devise a testing scheme that can detect sleep conflicts in device drivers with high probability.

We make several observations about the timing of smartphone power behavior. First, whenever the system is on, the CPU uses an RTC timer to retry system suspend every 20ms. The inactivity sleeping policy, *e.g.,* the system suspends after 15 seconds of user inactivity, is implemented by the kernel power manager holding a special wakelock for 15 seconds which we denote as the "inactivity" wakelock. Second, when a device driver transitions its device to the active power state, it either has invoked some sleep conflict prevention mechanism, or it has not. If it has not, this indicates a potential sleep conflict, as the system can suspend at any time. This condition can be checked by forcing the system to attempt the suspension process *right after* a device has entered the active power state, by releasing the inactivity wakelock.

The above observations motivate the design of the following testing scheme. For each device on the phone, we run a testing app that simply uses the proper API to trigger the device to enter the active power state, *e.g.,* writing to the sdcard, or vibrating the vibrator. As soon as the HYPNOS runtime observes the device has entered the active power state, it releases the inactivity wakelock, and the CPU will initiate the system suspend process. If HYPNOS's notifier is invoked and detects a potential sleep conflict, we know (1) none of the other drivers protected the system from suspension on behalf of this driver, and (2) the device driver did not use any mechanism to prevent sleep conflict. Hence, there is a sleep conflict bug in the driver. If the system suspend attempt does not succeed, *i.e.,*HYPNOS's notifier is not even invoked, it shows that there is currently some protection against system suspend. Then, the HYPNOS runtime system waits for up to $t_{util} + t_{timeout}$, where t_{util} is the time it would take the device to finish the workload in the testing API call, recorded during training, and $t_{timeout}$ is the timeout for the tail power state, if any. If HYPNOS' notifier is invoked and detects a potential sleep conflict before this point, there is a sleep conflict for the same reasoning as above. If not, since effectively the CPU has been on all along, the driver will correctly transition the device back to the suspend state.

Since after every failed system suspend attempt the CPU always stays up for 20ms, there is a vanishingly small chance that a short (*i.e.,* less than 20ms), unprotected time-critical section in the device drivers will go undetected by the above testing scheme. However, it is very unlikely such code sections exist, and if so actually give rise to a sleep conflict. When they do, HYPNOS's runtime avoidance scheme will detect and avoid the pending sleep conflict.

5.4 Discussion

To avoid sleep conflict, HYPNOS sometimes aborts suspension, and hence can potentially lead to increased CPU energy consumption. We justify this design decision as follows.

First, in Algorithm 1, HYPNOS allows suspension and uses an RTC to wake up the CPU for Case 2. For Cases 1 and 3, keeping CPU on ensures the progress of the app, which eventually will call APIs to turn off the device. If the CPU is off, the device would stay on in vein, since the app cannot run and hence make use of the device, *e.g.,* read the GPS. For Case 4, there is a potential tradeoff between a device draining power indefinitely versus the CPU draining extra power. The "optimal" decision is not easy to calculate due to the uncertainty of future external events, *e.g.,* when the user will press a key to wake up the CPU. We acknowledge HYPNOS mitigation can be refined to be more intelligent, *e.g.,* do not keep the CPU on for extremely low power devices such as sensor devices. We leave this refinement as future work.

6. Implementation

We implemented HYPNOS in Android 2.3 with the Linux kernel provided by CyanogenMod [2]. We implemented it in an extensible fashion so that developers can easily add checkers for new device drivers. The core sleep conflict detection engine entails 650 lines of code, and adding a checker for a new device typically adds around 50 lines of code.

The HYPNOS core exports the APIs for registering and unregistering FSM power models of the devices, `register_fsm (struct fsm*)` and `unregister_fsm(struct fsm*)`, respectively. Using the APIs, a new device can be registered with HYPNOS by adding a Linux module which can be applied dynamically, *i.e.,* the kernel need not be recompiled. The data structure `struct fsm*`, defined in the checker module and passed in the registration APIs, defines all the power states and transition types of the device. In `struct fsm*`, the checker also picks the bus driver-device driver interaction, from a list of interactions intercepted by HYPNOS, that it is interested in. It additionally provides a pointer to a callback function that has all the specs for the device, and walks the device power FSM according to the values passed to the lower-layer API (Section 5.2). This callback function is called whenever the HYPNOS core intercepts a bus driver-device driver interaction that the checker is interested in and passes it the intercepted parameters. After registration, the device is automatically tested for sleep conflicts in the HYPNOS suspend notifier.

Tracking interactions between device drivers and bus drivers. To make fixed APIs for device drivers to interact with bus drivers, the Linux kernel wraps bus driver APIs around standard APIs (for each bus type) and presents them to the device drivers. The bus drivers implement these APIs and register them with the kernel during initialization. Listing 4 shows an example of the GPIO wrapper function gpio_set_direction(unsigned gpio, int value). Each bus driver for each GPIO device registers struct gpio_chip with the kernel which contains the device-specific set function. The kernel's gpiolib converts the gpio_set_direction call to the device's specific set call. We log the device-specific calls in the wrapper function by inserting a call to our HYPNOS runtime system. We note that the function is made atomic by the surrounding spin locks (line 10), that prevent system suspend which could lead to a sleep conflict since the device has not been turned off yet (line 11).

Listing 4: An example of the kernel's wrapper function to provide uniform interface to device driver

```
1  struct gpio_chip{
2      void (*set)(struct gpio_chip *chip, unsigned
           offset, int value);
3    ..
4  }
5  void gpio_set_value(unsigned gpio, int value) {
6    struct gpio_chip  *chip;
7    chip = gpio_to_chip(gpio);
8    spin_lock_irqsave(&gpio_lock, flags);
9    // Added code for intercepting the call
10   hypnos_gpio_set_value(gpio, value);
11   chip->set(chip, gpio - chip->base, value);
12   spin_lock_irqrestore(&gpio_lock, flags);
13 }
```

Tracking hardware wakeups. If a device (*e.g.,* WiFi wake-on-lan) has the physical capability to wakeup the system, its device driver needs to register with the kernel to enable relevant hardware signals via explicit APIs such as `enable_irq_wake()` for GPIO signals, and `pci_enable_wake()` for PCI's PME# signal. The kernel power manager provides API `device_may_wakeup()` which can be used by HYPNOS to poll if a device is enabled to wake up the system.

7. Evaluation

We flashed HYPNOS with the kernel on Google Nexus One and HTC Magic phones running Android 2.3. Although we only tested HYPNOS on two phones, we have released the HYPNOS source code at `http://github.com/hypnos-android/`, which can be downloaded and flashed with the kernel on any phone running Android for testing all its devices.

7.1 Overhead Evaluation

The primary cost of using HYPNOS to perform online avoidance is the time spent in logging device and bus driver interactions, filtering out unwanted device driver-bus driver interactions, walking the FSM power model for each device in use, and running the avoidance algorithm at each successful system suspend, *i.e.,* after checking no wakelock is held and all suspend notifiers give permission to suspend. We measure the performance overhead of HYPNOS in a large data transfer via WiFi, which has the most intensive device and bus

driver interactions out of all devices. We run a simple app that sends 500KB, 1MB, 2MB, and 4MB, separately, to a remote server via WiFi and measure its completion time under a clean kernel and under one with the HYPNOS. We run each test three times and note the average. The completion time is 1.03%, 0.92%, 1.14%, 1.84% higher than the native performance, respectively. The small running time increase also serves as the upper-bound on the energy overhead, as apps may also consume I/O energy while HYPNOS does not. These results show that running HYPNOS online to perform sleep conflict avoidance, even for high-throughput devices, has very little cost. There is no successful system suspend during the above tests as the app was holding a wakelock. We then measured the overhead of HYPNOS in a successful system suspend at the end of the above test, after the wakelock was released. We repeated the measurement 10 times, and found that on average the original kernel takes 21.96 ms to complete the system suspend process and the kernel with HYPNOS takes 22.35 ms.

7.2 Sleep Conflict Testing Methodology

To systematically test all drivers of modern smartphone devices, we group smartphone devices into four categories according to how data is being transferred between the device and the CPU, as shown in Table 2.

- On/Off devices: This category consists of the simplest type of devices such as vibrator, LEDs, and flashlight. These devices just have two power states: on and off. They are typically attached to the general purpose input/output (GPIO) ports of the CPU which are configured as output ports.
- Input devices: This category includes all the sensors such as touch-screen, button, and accelerometer.
- Output devices: Speaker is an example output data device and is controlled via explicit program input.
- Two-way data transfer devices: This category consists of the more traditional I/O devices which transfer data between the CPU and the devices, such as sdcard, WiFi, 3G, and 4G. On smartphones, these devices all exhibit tail power states after exiting from their active power states, due to optimizations that try to minimize the startup time of future data transfers [14, 22].

A consequence of grouping devices according to the nature of data transfer to/from the CPU and operating modes, *e.g.,* explicit on/off versus implicit data oriented, is that the FSM power models of devices in the same category in Table 2 share similar power states and transition rules. Consequently, their device drivers which implement the transition rules are likely subject to the same or similar categories of sleep conflicts. Exceptions to the above, *e.g.,* some transitions for vibrator (Figure 1) and LED (Figure 4), both belonging to the On/Off category, differ, implying their drivers will experience different categories of sleep conflicts. Following the classification in Section 4, Table 2 lists the potential sleep conflicts for each category of devices. In the following, we discuss the nature of each device category and the detection results of HYPNOS.

Table 2: A taxonomy of major smartphone device drivers and their potential sleep conflict categories (defined in Table 1).

Device Category	Sleep Conflict Category			
	Case 1	Case 2	Case 3	Case 4
On/Off:				
vibrator		X		
LED, Flashlight			X	
Input (sensors, GPS):				
GPS			X	
proximity, compass			X	X
accelerometer, magnetic			X	
Output:				
speaker			X	
2-Way Data Transfer:				
sdcard	X	X	X	
WiFi, 3G, 4G	X	X	X	X

7.3 On/Off Devices

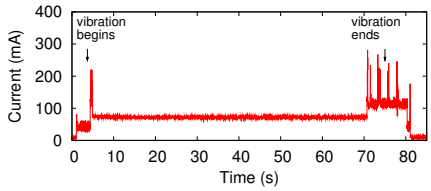
As discussed in Section 5.2, in Android, fixing the upper layer interface of device drivers is achieved via *device classes*, which provide uniform handling of functionally similar devices even if they are physically very different.

The *vibrator* is a part of the `timed_output` class in Android, and has been discussed in detail in Section 2. The multiple *LEDs* and *flashlight* belong to the `leds` class in Android. Each LED and each flashlight have separate `brightness` files in `sysfs`, and a user-space program can write any desired brightness from 0 to `max_brightness` of the LED in a respective file. Figure 4 shows the FSM of the LEDs. As with vibrator, both triggers are from user-space interactions.

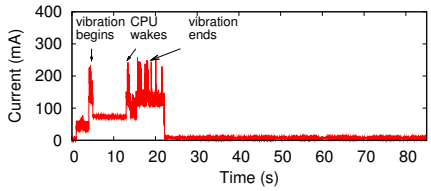
In summary, since on/off devices are explicitly turned off via API calls from user-space programs, either via a timer value for devices in the `timed_output` class or separate API calls for devices in the `leds` class, Cases 2 and 3 sleep conflicts can happen if a system suspend happens while waiting for the timeout or for program-input to explicitly turn off the device, respectively.

We tested the vibrator and LED drivers on the two phones, and HYPNOS detected Case 2 sleep conflict in both phones' vibrator drivers, but no sleep conflict in the LED drivers. Figure 8 shows the power drain of the phone, *i.e.,* the CPU and vibrator, with and without HYPNOS on the Nexus One phone while running the bug-inducing app (Listing 2).

We observe that with or without HYPNOS, the CPU was initially asleep at the start of the experiment. After about 2 seconds, the CPU wakes up (to possibly perform some background activity), causing a power spike, and continues the execution of our testing app. At the 5th second, the



(a) Vibrator sleep conflict.



(b) Sleep Conflict mitigation by HYPNOS.

Figure 8: Power consumption with and without HYPNOS.

vibration begins. The CPU then quickly suspends but the phone continues vibrating, draining about 70mA. Figure 8(a) shows without HYPNOS the vibration continued till the 72nd second, when the phone was explicitly powered on by the user. Figure 8(b) shows that HYPNOS correctly detected the sleep conflict when the CPU attempted to suspend at the 5th second, and set a wakeup RTC timer of 10 seconds. The CPU woke up at the 15th second and the vibration was properly stopped.

7.4 Input Devices

Input devices include touchscreen, GPS and all of the sensors such as proximity, magnetic, accelerometer, and compass sensors. All such devices belong to the `input` device class in Android.

Compared to other devices, the sensor devices are typically very low-power. For example, typical makes of the four types of sensors, proximity, magnetic, accelerometer and compass sensor consume about 45 μA, 5μA (for BU52014HFV from ROHM semiconductors), 290μA (for Bosch bma150) and 10μA (for AK8973), respectively. Consequently, many device drivers leave them always on. Alternatively, they can be explicitly turned on and off via API calls `registerListener(SensorEventListener l, Sensor s, int rate)` and `unregisterListener(SensorEventListener l)` on Android. In such usage, there can be a Case 3 sleep conflict if the system is suspended while the device is waiting for the turn-off API call.

Out of the four kinds of sensors, proximity and compass sensors are polling-based and accelerometer and magnetic sensors are interrupt-based, where the sensor hardware can go to low power by themselves and periodically wake up to provide readings via interrupt according to the frequency specified by parameter `rate`. In such cases, there can be a Case 4 sleep conflict if the system is suspended while such a sensor is still running and its device driver is waiting for more sensor interrupts before finally turning it off. We tested

Listing 5: Sleep conflict bug in the touchscreen driver of Google Nexus One phone.

```
1 static int synaptics_ts_suspend(struct i2c_client
      *client, pm_message_t mesg)
2 {
3   ...
4   ret = i2c_smbus_write_byte_data(client, 0xf0, 0
        x86); /* deep sleep */
5   if (ret < 0)
6     printk(KERN_ERR "synaptics_ts_suspend: failed\
          n");
7   ...
8   return 0;
9 }
```

sensor drivers on the two phones and HYPNOS identified sleep conflicts in AKM8973, the compass driver of Google Nexus One phone.

HYPNOS detected a sleep conflict bug in the touchscreen driver of Google Nexus One. When this bug occurs, the phone continues to draw 11.5mA of current after system suspension, as opposed to 3mA in the bug free case. Listing 5 shows the suspend notifier of the buggy touchscreen driver code. On line 4, the driver attempts to put the touchscreen device in the deep sleep mode. However, even when it detects that the device could not be put to sleep (line 5), the driver still returns 0 (line 8) from the suspend notifier, letting the CPU to suspend and hence causing sleep conflict.

GPS can consume substantial power, *e.g.,* 65mA on Google Nexus One. Therefore, it is always explicitly turned on and off via program input, and hence can suffer a Case 3 sleep conflict. However, on the two phones, HYPNOS did not detect any sleep conflict in their GPS drivers.

7.5 Two-Way Data Transfer Devices

Sleep conflict in this category of traditional I/O devices such as sdcard, WiFi NIC, 3G, 4G devices are more severe as these devices draw significant amount of power, and can continue draining power for a long period of time without being noticed by the user.

Due to its unique inner workings, an sdcard consumes between 20-100mA while reading/writing; reading typically consumes less power than writing. They support variable voltages ranging from 2.7 to 3.6V and variable clock rates from 100KHz to 50MHz. Additionally, they also support variable bus widths from 1 to 8 and different bandwidth.

The simplest sdcard controller drivers, such as CB710, do not change the voltage, clock frequency, or the bus width. Advanced sdcard controllers, such as `omap-hsmmc`, typically vary the voltage, clock frequency, bus width, and power states. The `omap-hsmmc` driver employs three separate timers of 100ms, 1s, and 8s to transition from enabled to disabled, sleep, and off state, respectively. In the disabled state, only the clock is turned off. In the sleeping state, the clock is off, and card and voltage regulators are asleep. In the off state, the voltage regulator, card and clock are all off.

Sleep conflicts can arise in using an sdcard because it involves many power states (enabled, disabled, asleep and off) and transitions. Since sdcards may involve active utilization for long periods of time (*e.g.,* large data transfer), the system may suspend during the active utilization period (Case 1). Sleep conflicts can also happen while waiting for timeouts (Case 2) to switch to the disabled, asleep, and off states. Further, program inputs such as file close can cause the sdcard driver to immediately transition the sdcard from enabled to off and hence a system suspend that prevents any program input would lead to a Case 3 sleep conflict. We tested the sdcard drivers on the two phones, and HYPNOS did not detect any sleep conflict.

The power model for WiFi on the HTC Magic phone in Figure 5 shows the WiFi NIC transitions among a base state, an active state, and a tail state. Cases 1, 2, 3, and 4 sleep conflicts can happen if the system suspend happens while in active use, waiting in the tail state for the timeout, waiting for program-input to explicitly turn off the device, or waiting for incoming packets but the wake-on-lan was mistakenly not enabled, respectively. We tested WiFi NIC drivers on the two phones, and found a sleep conflict in the WiFi NIC on HTC magic. After a burst of WiFi transmissions, the driver puts the WiFi NIC to the tail energy state and intends to put it to the off state after an inactivity timeout of 1.5 seconds. A system suspend sleeps during this tail state triggers a Case 2 sleep conflict, as the NIC driver did not use an RTC timer to wake up the system at the end of timeout.

8. Related Work

Energy Debugging in Smartphones:. Energy bugs in smartphone apps were first discussed in [17], which developed a taxonomy of energy bugs. ARO [21] and eprof [18] are two energy debugging tools which provide hints to developers on how to reduce the app's energy consumption. Recently, [20] characterizes and detects no-sleep energy bugs caused by not releasing wakelocks in smartphone apps. This technique is based on a reaching definitions dataflow analysis. In contrast, our work investigates a class of new energy bugs, sleep conflict, that occur in smartphone device drivers, and proposes automatic techniques for runtime avoidance and for pre-deployment testing of these bugs. Carat [16] performs collaborative debugging to identify "energy hog" apps via statistical inference from the observed behavior of an app running on many phones.

Device Driver Debugging:. Device drivers have been reported to account for a majority of system crashes. The large body of work on improving the safety and reliability of device drivers fall into two broad categories: offline testing and online mitigation. Among offline approaches, static analysis tools (*e.g.,* [1, 3, 10, 27]) are able to quickly find specific bugs such as misuse of the kernel and drivers, but have difficulties with code features such as pointers and multiple invocations of the driver. Formal specification first expresses a device or driver's operational requirement which allows other parts of the system to verify the correct operation of drivers (*e.g.,* [11, 29]). However, modern OS device driver architectures are deemed too complicated to be formally specified [8]. Recently, several works have applied symbolic execution to testing device drivers [12, 15, 24]. SymDrive [24] further combines static analysis with symbolic execution to reduce the effort of testing a new driver. Among online mitigation approaches, Nooks [26], SFI [28] and XFI [13] use lightweight protection domains or software isolation techniques to provide fine-grained isolation of drivers. Finally, Shadowdrivers [25] conceal driver failures from the OS and applications and transparently restore the drivers back to functioning states. In contrast to the above work, we present the first study of a class of energy bugs, sleep conflict, in smartphone OS device drivers. These bugs do not cause system crashes, but can lead to unexpected, significant battery drain due to devices necessarily staying in active power states. We further developed HYPNOS, the first runtime system that treats sleep conflict bugs.

Sleep-Cycle Management in Other OSes. To our knowledge, Linux and RTOSes also use mechanisms similar to suspend notifiers to notify the drivers that the system is about to suspend (*e.g.,* [5, 6]). Further, many RTOSes, *e.g.,* Nucleus OS and Windows CE, expose mechanisms similar to Android wakelocks to their applications [4, 7]. However, to our best knowledge, there has been no work on energy bugs in general, and sleep conflict bugs in particular, in the RTOS and Linux literature. We believe sleep conflict can also happen in RTOSes and Linux and expect our detection and mitigation to be applicable there as well.

9. Conclusion

This paper presents the first study of a new class of energy bugs, sleep conflicts, which can occur in smartphone device drivers. We describe the root cause of sleep conflicts, develop a classification of sleep conflicts according to the transitions in the power models of devices, and present HYPNOS, a runtime system that performs pre-deployment testing and sleep conflict avoidance in deployed systems. On two Android phones, HYPNOS detects several sleep conflict bugs and prevents them from draining the battery. HYPNOS does not require access to driver source code, making it widely applicable. We have released HYPNOS at `http://github.com/hypnos-android` and envision it being used by smartphone OS/driver developers to test drivers for sleep conflict bugs and to protect Android users from mysterious battery drain due to sleep conflicts.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Pablo Rodriguez, for their helpful comments. This work was supported in part by NSF grant CCF-0916901.

References

[1] Coverity. Analysis of the Linux kernel, 2004. http://www.coverity.com.

[2] Cyanogenmod: Android community rom based on froyo. http://www.cyanogenmod.com/.

[3] Microsoft Corporation. static driver verifier, May 2010. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx.

[4] Nucleus power management. http://www.mentor.com/embedded-software/nucleus/power-management.

[5] Suspend and resume handling in windows CE. http://msdn.microsoft.com/en-us/library/ee497733.aspx.

[6] Suspend notifiers in linux. http://www.kernel.org/doc/Documentation/power/notifiers.txt.

[7] Windows CE power management. http://msdn.microsoft.com/en-us/library/aa917915.aspx.

[8] S. Amani, L. Ryzhyk, A. Donaldson, G. Heiser, A. Legg, and Y. Zhu. Static analysis of device drivers: We can do better! In *Proc. of 2nd APSYS Workshop.*, 2011.

[9] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc of IMC*, 2009.

[10] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *Proc. of POPL*, 2002.

[11] M. Barnett, M. Fahndrich, K. R. M. Leino, P. Muller, W. Schulte, and H. Venter. Specification and verification: The spec# experience. *Commun. of the ACM*, 54(1), June 2011.

[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proc. of ASPLOS*, 2011.

[13] . Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. of OSDI*, 2006.

[14] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *Mobisys*, 2012.

[15] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with ddt. In *Proc. of USENIX ATC*, 2010.

[16] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica. Collaborative energy debugging for mobile devices. In *Proc. of USENIX HotDep*, 2012.

[17] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging for smartphones: A first look at energy bugs in mobile devices. In *Proc. of Hotnets*, 2011.

[18] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proc. of EuroSys*, 2012.

[19] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system-call tracing. In *Proc. of EuroSys*, 2011.

[20] A. Pathak, A. Jindal, Y. C. Hu, and S. Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of Mobisys*, 2012.

[21] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proc. of Mobisys*, 2011.

[22] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing radio resource allocation for 3g networks. In *Proc. of IMC*, 2010.

[23] R. Muralidhar et al. Experiences with power management enabling on the Intel medfield phone. In *Proc. of Linux Symposium*, July 2012.

[24] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing drivers without deevices. In *Proc. of OSDI*, 2012.

[25] M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. of OSDI*, 2004.

[26] M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), Feb. 2005.

[27] T. Ball, et al. Thorough static analysis of device drivers. In *Proc. of EuroSys*, 2006.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of SOSP*, 1993.

[29] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of OSDI*, 2008.