Unsafe Time Handling in Smartphones

Abhilash Jindal, Prahlad Joshi, Y. Charlie Hu, and Samuel Midkiff, *Purdue University*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/jindal

This paper is included in the Proceedings of the
2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Unsafe Time Handling in Smartphones

Abhilash Jindal
Purdue University

Prahlad Joshi
Purdue University

Y. Charlie Hu
Purdue University

Samuel Midkiff
Purdue University

Abstract

Time manipulation, typically done using gettime() and settime(), happens extensively across all software layers in smartphones, from the kernel, to the framework, to millions of apps. This paper presents the first study of a new class of software bugs on smartphones called sleep-induced time bugs (SITB). SITB happens when the phone is suspended, due to the aggressive sleeping policy adopted in smartphones, in the middle of a time critical section where time is being manipulated and delay caused by unexpected phone suspension alters the intended program behavior.

We first characterize time usages in the Android kernel, framework, and 978 apps into four categories and study their vulnerabilities to system suspension. Our study shows time manipulation happens extensively in all three software layers, totaling 1047, 1737 and 7798 times, respectively, and all four usage patterns are vulnerable to SITBs. We then present a tool called KLOCK, that makes use of a set of static analyses to systematically identify sleep-induced time bugs in three of the four time usage categories. When applied to five different Android Linux kernels, KLOCK correctly flagged 63 SITB-vulnerable time manipulation instances as time bugs.

1 Introduction

Smartphones designs are increasingly subject to three diametrically opposed constraints: phones must have increasing software and hardware functionality which can increase power requirements; phones have limited form factor and weight which bounds the size of their battery and therefore their power supply; and phones must have ever increasing battery life to meet user expectations and be competitive in the market place. The push for maximal energy savings under these constraints quickly drove their OSes, such as Android, to pursue an aggressive system sleeping policy. After some set period of user inactivity, *i.e.,* the user has not touched the screen or any peripheral buttons on the device, the OS power management triggers the phone's system on chip (SOC) to enter its default state, *system suspend*, where all components on the SOC are suspended, RAM is put in a self-refresh mode, and the SOC drains close-to-zero battery power.

The difficulty with this approach is that applications, the application framework (supplied by the OS vendor and providing low-level services to the apps) and the kernel (which implements many of the low-level services provided by the framework) often have *time critical sections* that are not part of interactive code. A time critical section is a dynamic code region (*i.e.,* code that may not be textually contiguous but is logically related) over which the system should not suspend.

To prevent the system from suspending when performing time critical work, smartphone OSes have exported mechanisms to allow programmers to prevent system suspension in selected parts of the program. Primary among these mechanisms is the *wakelock*, which, when acquired, prevents the system from suspending and when released allows it to suspend if nothing else is preventing the suspension.

However, mobile phone kernels and apps are complex. They utilize an event driven programming model and are often concurrent. Combining this complicated programming model with explicitly managing the SOC suspend/awake cycle unavoidably results in *sleep disorder bugs*, *i.e.,* programming mistakes in system suspension management that result in unintended app or hardware device behavior.

Pathak *et al.* [25] presented the first study of sleep disorder bugs, focusing on a class of bugs that result from not releasing wakelocks in apps, preventing the phone from going to sleep and draining energy. Jindal *et al.* [16] studied sleep conflicts, another class of sleep bugs that happen in device drivers where a phone's component (*e.g.,* WiFi) is left in a high power state wasting battery. Sleep conflicts occur when a component in high power state is unable to transition back to its base power state because the system got suspended before the device driver responsible for driving the power transition could run. In both studies, the sleep bugs targeted do not lead to incorrect program behavior; instead they cause excessive battery drain due to preventing the SOC/CPU or I/O devices from going to sleep.

This paper studies a new class of sleep-related bugs, called *sleep-induced time bugs*. Sleep-induced time bugs can occur at all levels of the mobile phone ecosystem – in apps, the framework and the OS code. Unlike previously studied sleep bugs which lead to energy leaks, sleep-induced time bugs manifest as logical errors resulting from time values becoming "stale" because the CPU sleeps during the manipulation of time related values. The manipulations occur via `gettime()`, `settime()`, time arithmetic APIs, and simple arithmetic operations, and the new class of bugs manifest themselves as incorrect values in program variables rather than an incorrect power state of the device hardware.

Sleep-induced time bugs are difficult to reproduce and debug since they only arise during some particular, intricate timing between time-critical section execution and CPU suspension. Indeed when we submitted a patch to the bug we found in the DHT11 humidity and temperature sensor driver, the kernel maintainer responded [8] by saying:

> "I think it will fix an odd issue I have seen in a log file (apparently was completely off track debugging it). As this very likely is a real world issue, I'd recommend applying the patch to the fixes branch [sic]."

In this paper, we make three contributions towards understanding and treating sleep-induced time bugs.

First, we characterize time manipulation usages and their vulnerabilities to system suspension in the Android kernel, framework, and 978 apps. We find time manipulation happens extensively in all three software layers, totaling 1047, 1737 and 7798 times, respectively. We further classify all time usages into four categories: timed callbacks, setting the time, time arithmetic, and timed logging. The categorization uncovers the time critical sections in each category, their vulnerability to sleep-induced time bugs, as well their syntactic characteristics which give hints for detecting them.

Second, to allow programmers to isolate and fix sleep-induced time bugs, we present a tool called KLOCK that detects all instances of the first three categories of time manipulations in the Linux kernel. KLOCK exploits a key observation that the start and/or end of time critical sections due to time usage in the three categories are marked with a handful of APIs that get/set time or register timer callbacks in the Linux kernel. KLOCK makes use of a collection of sophisticated compiler analyses – Use-Def and Def-Use chains, points-to analysis and reaching definitions analysis as building blocks, and customizes their integration to detect time bugs in each of the three categories of time usages.

Third, we have implemented KLOCK and applied it to five Android Linux kernel versions. KLOCK aided in detecting 63 time bugs, out of which, we found 14 have been fixed and 7 files with bugs have been removed from later versions of the kernel. We reported the remaining 42 bugs to the corresponding Linux kernel mailing lists, and out of the 7 developers who have replied so far, all confirmed the reported bugs and accepted our patches.

Although we focus on the Android Linux kernel in this paper, we believe KLOCK is quite general and its design can be applied to detect sleep-induced time bugs in the framework, the apps, and other software systems that are vulnerable to system suspension, the de facto technique for energy saving on mobile systems.

2 Background

We start with a brief overview of the system suspension process aggressively triggered on modern smartphones and the complex set of clock options provided by the Linux kernel.

2.1 System Suspension in Smartphones

A modern smartphone consists of the SOC and numerous hardware I/O devices such as LCD, SD Card, WiFi NIC, cellular, GPS, cameras, and accelerometer. The SOC consists of the CPU, RAM, ROM, and micro-controller circuits for various phone devices such as GPS, graphics, video and audio. The default SOC power state is suspend, where all components on the SOC are suspended, RAM is put in a self-refresh mode, and the SOC drains close-to-zero battery power.

Wakelocks are a type of explicit power control APIs with two associated API calls, acquire and release. Wakelocks are also exported to the user space to support background services as well as non-interactive foreground jobs. The Android framework, apps, and device drivers extensively use wakelocks to ensure continuous execution of code sections.

When the last wakelock is released, the wakelock kernel module immediately attempts system suspension, by calling `pm_suspend()` to perform four tasks serially. First, the filesystem is synced by moving the buffered data from RAM to NAND. Second, all the user processes and kernel threads are frozen. Third, it attempts to suspend devices by calling the list of *suspend callbacks* registered by device drivers which power down their respective devices. Note that any suspend callback may return failure because it is waiting on a condition variable which is set to false elsewhere in the kernel, which would abort the entire suspend process. Finally, all the CPU cores are disabled by calling the architecture specific code to complete the suspension.

Note that system suspension is only attempted when the last wakelock is released. If interrupts in the system are disabled, the running process cannot be context switched to another process that might release the wakelock or get interrupted by wakelock timer expiration. Thus disabling interrupts in a code section effectively prevents suspension.

In summary, system suspension will not succeed while a piece of code is executing if it (1) holds a wakelock; (2) disables interrupts indirectly preventing the last wakelock from getting released; or (3) sets a condition variable that causes a suspend callback to return failure and hence abort any suspension attempt.

2.2 Timekeeping in Linux

The Linux timekeeping subsystem is responsible for maintaining and providing current time to the rest of the kernel. The POSIX standard requires the timekeeping subsystem to maintain `CLOCK_REALTIME` which is the time elapsed since the midnight of January 1, 1970. `CLOCK_REALTIME` is first read from the real time clock during the kernel initialization phase and then later updated at every tick.

However, `CLOCK_REALTIME` is susceptible to sudden changes due to the user setting the time or from ntpd, making it particularly unsuitable to measure elapsed time of a code section. To overcome this, the POSIX standard mandates the timekeeping subsystem to provide `CLOCK_BOOTTIME` which gives the time elapsed since the boot time. `CLOCK_BOOTTIME` can not be set by the user or by ntpd and hence does not suffer from sudden discontinuities like `CLOCK_REALTIME`.

But `CLOCK_BOOTTIME` is not quite suitable for measuring code execution time, because it includes the time elapsed even while the SOC is suspended. For this reason, the POSIX standard introduced `CLOCK_MONOTONIC` which works like `CLOCK_BOOTTIME` but pauses during SOC suspension making it suitable for measuring program execution time.

Although `CLOCK_BOOTTIME` and `CLOCK_MONOTONIC` will not be reset to suddenly jump backward or forward, their rate is still adjusted slightly to fix clock drifts which is done autonomously by the timekeeping subsystem. For this reason, in addition to POSIX standards, the Linux timekeeping subsystem also provides `CLOCK_MONOTONIC_RAW` which is simply the local oscillator not disciplined by NTP, for use in cases where more accurate time is needed over very short intervals.

In summary, the myriad of clocks available in the Linux kernel and their subtle semantics pose a significant challenge to the developers, and using the wrong clock leads to vulnerabilities to unexpected events such as system suspension.

Listing 1: Sleep induced time bug in Linux kernel memcpy benchmark: system suspend can alter the time arithmetic result.

```
1 double do_memcpy_gettimeofday(memcpy_t fn, size_t
       len...) {
2   struct timeval tv_start, tv_end, tv_diff;
3   alloc_mem(&src, &dst, len);
4   gettimeofday(&tv_start, NULL);
5   fn(dst, src, len);
6   gettimeofday(&tv_end, NULL);
7   timersub(&tv_end, &tv_start, &tv_diff);
8   return len / timeval2double(&tv_diff);
9 }
```

3 Sleep-Induced Time Bugs

Time manipulation occurs frequently across all layers of smartphone software, from the kernel, to the framework, to the apps. Two factors together give rise to *sleep-induced time bugs* (SITB). First, the smartphone OS employs an aggressive system suspend policy. Second, time manipulation in smartphone software forms a *time critical section* (TICS) whose start and end are marked by time manipulation APIs or operations involving values obtained from the time manipulation APIs. Any delay within the TICS caused by the smartphone suspension will alter the intended program behavior and give rise to an SITB. More formally, an SITB happens when the smartphone CPU/SOC is suspended in the middle of a TICS that alters the intended program behavior. We discuss the impact of these bugs in Section 4.

We now illustrate a sleep-induced time bug in the Linux kernel memcpy benchmark, `/tools/perf/bench/mem-memcpy.c`. The benchmark code measures how much time each of the various memcpy functions takes to copy a single byte. A code snippet is shown in Listing 1. The function accepts a pointer to the function `fn` being benchmarked and the length of the memory block to be copied. Before calling `fn` to start copying in Line 5, the current time is read into variable `tv_start`. After `fn` returns, the current time is read in variable `tv_end` in Line 6. Line 7 computes the time taken by `fn` by computing the difference between `tv_start` and `tv_end`. Line 8 then calculates the rate of copying by dividing `len` by `time_diff`.

Consider the scenario where the CPU sleeps in between the two calls of `gettimeofday`, in or outside `fn`, `tv_start` is set to T1 and `tv_end` is set to T4, but the system was suspended from T2 until T3. The code will incorrectly compute the time taken by `fn` as (T4 - T1), while the actual time taken is (T2 - T1) + (T4 - T3), and return an erroneous copying rate.

Table 1: **Time usage in the Anroid kernel, framework, and 978 apps.**

Usage Pattern	Static Use Count			Example Usage in Kernel
	Kernel	Android	App	
Timer Callback	477	215	352	kernel/time/alarmtimer.c, fs/timerfd.c
Time Setting	17	8	1	kernel/time.c, drivers/rtc/alarm.c
Time Arithmetic (lower bound)	125	522	236	net/ipv4/tcp_probe.c, kernel/time/tick-sched.c drivers/cpuidle/cpuidle.c fs/jbd/transaction.c
Logging (upper bound)	453	992	7209	fs/dlm/ast.c, net/ wireless/mwifiex/cmdevt.c net/sunrpc/svcsock.c
Total	1072	1737	7798	

4 Characterizing Time Usage and Vulnerability to Sleep-induced Time Bugs

To understand the prevalence of time usage, typical time usage patterns, and their vulnerability to sleep-induced time bugs in smartphones, we examined and classified all the time usage in the Android kernel, the framework, and a set of 978 apps. The classification gave us many insights into the root causes of SITBs and hints on how to detect them.

4.1 Time Usage in the Android Ecosystem

As discussed in §2.2, the Linux timekeeping subsystem provides a myriad of different clocks and exports the APIs (except for `clock_monotonic_raw`) at every software layer, from device drivers all the way up to apps. We first read the API documentation to collect the list of such time APIs exposed at each software layer [6, 5, 4, 1, 2]. We then grepped for all the usages of respective APIs in the source code of the kernel, the Android framework, and a set of 978 apps which included the 100 most popular apps on Google Play which we manually downloaded and 878 apps we crawled the day before Android Market was switched to Google play. The app source code were obtained by decompiling the apk files using ded [3].

Table 1 shows that time manipulation is prevalent in the Android ecosystem, totalling 1072, 1737 and 7798 times in the Android kernel, framework, and 978 apps.

4.2 Categorizing Time Usage and Vulnerability

To understand the purposes of time manipulation widely used in the smartphone software layers, we manually inspected 50 time usages found in each software layer and found them to fall into the following four categories. Understanding the usage of each category in turn allows us

Listing 2: Code that generates waveform using timed callback.

```
1  // Generates a sawtooth wave on channel 0,
     square wave on channel 1
2  static void waveform_ai_interrupt(unsigned long
        arg) {
3    do_gettimeofday(&now);
4    elapsed_time = USEC_PER_SEC*(now.tv_sec-
         devpriv->last.tv_sec)+(now.tv_usec-
         devpriv->last.tv_usec);
5    devpriv->last = now;
6    num_scans = (devpriv->usec_remainder +
         elapsed_time) / devpriv->scan_period;
7    for (i = 0; i < num_scans; i++) {
8      sample = fake_waveform(dev, ...);
9      cfc_write_to_buffer(dev->read_subdev,sample
           );
10   }
11   devpriv->usec_current += elapsed_time;
12   mod_timer(&devpriv->timer, jiffies + 1);
13 }
14 static int waveform_attach(struct comedi_device
        *dev, struct comedi_devconfig *it) {
15   ..
16   init_timer(&(devpriv->timer));
17   devpriv->timer.function =
         waveform_ai_interrupt;
18 }
```

to automatically search for all the instances of each usage, as explained below.

4.2.1 Case 1: Timed Callback

Usage Pattern. In this category, the code wishes to perform a certain task at a future time. The code registers an alarm with the system specifying the function that should be called and the time interval after which the callback should happen.

Listing 2 shows an example of how timed callbacks are set up and used in the kernel. The code generates waveforms of preconfigured shape. At driver initialization, function `waveform_attach` (line 14) is called which sets the pointer of the timed callback function `devpriv->timer.function`. The function `waveform_ai_interrupt` generates one period wide wave (lines 7-10) then recursively invokes itself via a timer callback (line 12).

Vulnerability. In this category of time usage, the duration from timer registration till the timeout (*i.e.,* when callback is supposed to be invoked) forms a time critical section. A time bug can arise when the CPU suspension happens in the middle of the TICS which delays the callback until the next time the CPU wakes up.

The waveform generation code in listing 2 contains a time bug. Since the driver does not protect against SOC suspension, the SOC might suspend after the timer is set at line 12 and cause large gaps in the waveform distorting its shape.

However, we observed not all such delays give rise to time bugs: time bugs arise only when the callback execution interacts with a peripheral I/O components. For

Listing 3: Clock synchronization in GsmServiceStateTracker.java class in Android (simplified for illustration).

```
 1 wakelock.acquire();     //Keep the CPU on
 2 x = gettime();          //Obtain external time
 3 if (a condition){     //not based on x
 4   /* lots of code */
 5   x = f(x, z);        //Fix x using z
 6   /* more code */
 7 } else {
 8   /* some code */
 9   wakelock.release(); //Release wakelock
10   /* lots of code */
11   wakelock.acquire(); //Re-acquire wakelock
12 }
13 settime(x);             //Set hardware time
14 wakelock.release();     //Release wakelock
```

Listing 4: Speed calculation in SpeedTest.net which measures the network connection speed.

```
1 protected Integer doInBackground(URL[]  r1) {
2   mStartTime = SystemClock.uptimeMillis();
3   //upload data
4 }
5 protected int getProgress(int  i0) {
6   //compute speed
7   i33 = i0 / (SystemClock.uptimeMillis() -
         mStartTime) / 1000;
8 }
```

example, an alarm app that wishes to ring an alarm at a user-specified time must ensure that the callback is processed at the intended time. On the other hand, the process scheduler in the kernel also registers a timer callback in order to schedule a new process at the end of a time slice, but even if the CPU suspends in the middle, the delay in callback execution will have no impact on the scheduler semantics.

Occurrences. To count the occurrences of this type of time usage, we compiled a list of all callback registration APIs that are exported at each software layer by reading the documentation [6, 2], and counted the number of occurrences of those APIs in the source code. Table 1 shows that the time callback is widely used, for a total of 477, 215, and 352 times in the Linux kernel, the framework, and the 978 apps, respectively.

4.2.2 Case 2: Time Setting

Usage Pattern. In this category, the subject code updates the current system time. For example, code listing 3 shows an excerpt from GsmServiceStateTracker in the Android framework that obtains the external time (line 2), performs manipulation over it (lines 3-12), and sets the local time (line 13).

Vulnerability. In this category, the duration from gettime() to settime() forms a time critical section. A time bug will arise when the CPU suspends in the middle of the TICS which causes the new hardware time set to be incorrect. For example, code listing 3 sets the time (line 13) obtained from the network (line 2). But at line 9, the programmer mistakenly releases the wakelock, giving an opportunity for the CPU to suspend between line 9 and line 11. When this happens, line 13 will run after the next CPU wakeup, setting a stale time.

Occurrences. To count the occurrences of this type of time usage, we compiled a list of exported APIs at the three software layers for setting the current system time and searched for them in the source code. We found 17, 8, and 1 instances of this type of time usage in the kernel, framework and the 978 apps, respectively, as shown in Table 1.

4.2.3 Case 3: Time Arithmetic

Usage Pattern. Another common time usage pattern is to collect two timestamps and perform arithmetic over them. The arithmetic is performed either directly via integer or long arithmetic, or using Linux provided helper functions dedicated to performing time arithmetic, *e.g.,* `timespec_sub`.

Code listing 4 is extracted from the SpeedTest.net app [9] which measures the speed of network connection by registering two callback functions with the framework. The first callback takes a timestamp and saves it in `mStartTime` (line 2), and uploads data to the test server (line 3). The second callback then computes the speed by finding the elapsed time by subtracting `mStartTime` from the current time (line 7).

Vulnerability. In this category, the duration between the actions of getting two timestamps forms a time critical section. Time arithmetic programs are vulnerable to two kinds of vulnerability.

(a) Due to system suspension. A time bug will arise when the CPU suspends in the middle of the TICS and the time arithmetic will output an incorrect value. For example, in code listing 4, the TICS which starts at line 2 and ends at line 7 is not protected by any wakelock. As a result, the CPU may suspend before line 7, and the computed speed will be much lower than the actual one.

(b) Due to resetting the time. Time bugs will also arise in time arithmetic when the user or ntpd resets the current time between the actions of obtaining two timestamps.[1] In addition to causing elongated elapsed time (as in system suspension vulnerability (a)), this vulnerability can cause elapsed time to elongate or shrink or even become negative since the time can be set to either future or past timestamps.

We note using `CLOCK_MONOTONIC` which ignores system sleep time and cannot be set by ntpd or user, as discussed in §2.2, would have avoided time bugs in such time arithmetic. However, due to their subtle semantics,

[1]We note this bug scenario can also arise in desktop/server platforms.

many kernel and app programmers make mistakes in using the gettime APIs.

Occurrences. To count the occurrences of this type of time usage, we searched for all the helper APIs that perform time arithmetic at the three software layers. Since time values may also be manipulated with direct integer/long arithmetic, the number of occurrences counted this way will be an underestimate. Table 1 shows that time arithmetic is extensively performed, with lower bounds of 125, 522, and 236 occurrences in the kernel, framework, and 978 apps, respectively. We note the SITB detection tool we present in §5, however, will capture all Case 3 bugs, whether the time is manipulated using arithmetic APIs or direct arithmetic.

4.2.4 Case 4: Logging

Usage Pattern. In this category, the code obtains the current time and logs it in conjunction with some event, usually for postmortem debugging.

Vulnerability. For such usages, the code between an event and its timestamping forms a time-critical section, as a CPU suspension in between will result in an incorrect timestamp being logged for the event. However, automatically detecting this category of TICSes is challenging, since in general there is no syntactic clue correlating the event and logging. We leave detecting SITBs in this category as future work.

Occurrences. Since it is difficult to count all such usages by searching any APIs, we heuristically assume that if a timestamp call, *i.e.*, gettime(), is not used in one of the above three categories, it belongs to this category. Hence the numbers for Case 4 time usage pattern in Table 1 are overestimates. Table 1 shows timed logging occur 453, 992, and 7209 times in the three software layers.

5 Design

Sleep-induced time bugs occur when a part of a time critical section (TICS) is unprotected. In this section, we explore the design space and present a detection system called KLOCK that automatically finds SITBs in the first three categories of time usages using static analysis.

5.1 Design Space

We explored the use of both software model checking and dataflow analysis, two primary techniques that have been extensively applied to finding software bugs [12, 21, 14, 28, 13, 19]. Both techniques attempt to discover properties that hold for the program or at certain points in the program.

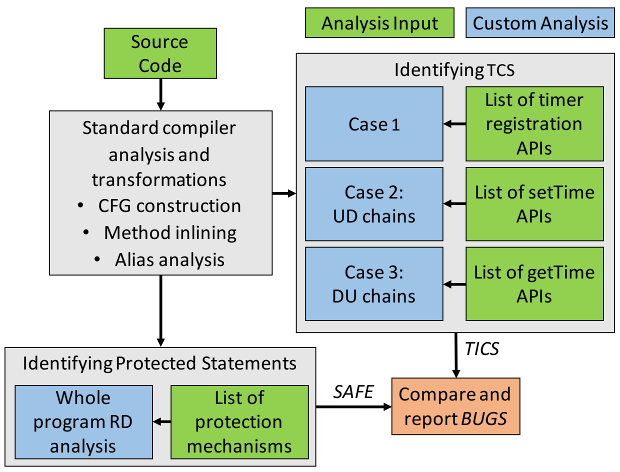

Figure 1: The KLOCK architecture.

Roughly speaking, model checking is well suited to analyses that explore dynamic properties of the system, and data flow analyses are typically used in situations where conservative approximations can be made regarding paths or inter-thread ordering. Which kind of analysis to choose depends on the implementation difficulty and the analysis precision that can be achieved for the property under consideration. As shown in [15], model checking is not always more accurate than data flow analysis and can be more difficult to implement. In the current work, the properties to be modeled (Use/Def information, program paths that may not be covered by a wakelock) corresponded well to what could be accomplished via traditional dataflow analysis. This, combined with the relative ease of implementing these within the LLVM framework led us to use dataflow analysis for our debugging solution.

5.2 Design Overview

Figure 1 gives an overview of the KLOCK design, and the overall detection algorithm. KLOCK takes as input source code and performs two major tasks to detect time bugs. First, it identifies each of the three types of time critical sections of interest by identifying the statements that delimit the TICS and the statements that are contained in the TICS (§5.4). Next, it finds the set *SAFE* of all *safe* statements, *i.e.*, statements that are safe from CPU suspension during their execution (§5.5). The statements that belong to *TICS* and not to *SAFE* are in a TICS that are subject to SOC/CPU suspension while they are executing; they are marked as sleep-induced time bugs and added to the set *BUGS* which are reported.

5.3 Compiler Analyses Used by KLOCK

Our system uses a number of well-known compiler analysis techniques [10]. These include (1) **Points-to anal-**

ysis determines what can be pointed-to by a pointer or reference. Our techniques will use points-to analysis to find the targets of function pointers. (2) **Interprocedural reaching definitions analysis** (RDA) finds all definitions of variable v that can reach a statement that uses variable v. (3) **Use-Def and Def-Use chain construction** links together definitions of variables and uses of those definitions across functions within the program. Use-Def and Def-Use chains are used to follow the flow of data through the program and form the core of our analysis.

5.4 Identifying Time Critical Sections

Identifying time critical sections is, in general, impossible because whether or not a set of statements is a time critical section depends on the intended semantics of the code. We make a key observation that in all three categories of time usage that are vulnerable to time bugs, the start and/or end of time critical sections are marked with a handful of APIs that get/set time or register timer callbacks in the Linux kernel. Therefore, by identifying and informing the compiler of these APIs, *e.g.,* by providing them in a table to the compiler, we can effectively bootstrap the compiler analysis for precisely identifying all such time-critical code sections.

Case 1: Timer Callback. Recall that in this case, the program registers a timer callback for performing a time critical task. The TICS contains the registration, the callback and the critical task performed by the callback.

Algorithm 1 TICS identification for Case 1: Callbacks

```
Require: Program $P$, Callgraph $C$, Timer registration APIs set $R$
1: Time critical statements $TICS \leftarrow \emptyset$
2: for all Statements $S_P \in P$ do
3:     if $S_P$ calls function $F_R \in R$ then
4:         $TICS = TICS \cup S_P$
5:         Callback $F_C$ = getTarget($S_P$,$C$)
6:         $TICS = TICS \cup F_C$
7:     end if
8: end for
9: return $TICS$
```

Algorithm 1 gives the pseudo code for detecting callback based *TICS*. To identify the start of a TICS, we identify the small number of callback registration functions R, such as `hrtimer_start` which registers a high-resolution timer in the kernel. Identifying the start of a TICS boils down to matching all calls in the source code against the list of functions in R (line 3).

The next step is to correctly identify the callback function corresponding to every timer registration (line 5). In the kernel, registration is done by passing a pre-defined struct that contains the callback function pointer. For example, in code listing 2, the struct `devpriv->timer` is passed as an argument to `mod_timer` in line 12, and its member `.function` is set to `waveform_ai_interrupt` in line 17. As shown in this example, the callback registration (line 12) and setting function pointer (or defining argument object) can be in disconnected places. Hence, identifying the correct callback function in the kernel code requires pointer analysis.

The end of a TICS should, ideally, be marked by identifying the end of time critical processing inside the timer callback. Since identifying this critical processing can be highly context sensitive, we make a conservative assumption that the TICS ends when the timer callback exits. Hence, we add all the statements in the timer callback to *TICS*, in Algorithm 1.

Case 2: Time Setting. Recall that in this case, the TICS ends with a function call that sets the clock and begins at the point when the time variable used to set the clock was first obtained. Because there are only a small number of APIs that set the clock (*e.g.,* `settimeofday`), a list of them can be maintained in the compiler, and a call to one of these will mark the end of a TICS.

Algorithm 2 TICS identification for Case 2: Set time

```
Require: Program $P$, Callgraph $C$
Require: Statement-variable tuples $(S_{ST}, V_t)$ that set time
1: Time critical statements $TICS \leftarrow \emptyset$
2: for all $(S_{ST}, V_t)$ do
3:     $TICS \leftarrow TICS \cup DEFS^{+}(S_{ST}, V_t)$
4: end for
5: return $TICS$
```

Identifying the start of *TICS*, however, is more complicated. This is because the time value that is used to set the clock can be read from the network (*e.g.,* via NTP) or can be directly obtained from the user, *i.e.,* there is no fixed API for obtaining this time value. We do know, however, that the variable obtained from other sources must affect the time variable used to set the clock. Hence, to find the start of Case 2 TICS, we find the transitive closure $DEFS^{+}(S_{ST}, V_t)$ of the use-def chain ($DEFS$ set) where S_{ST} is a statement with an API call to set time and V_t is the variable containing the time for that call. $DEFS(S,V)$ is the set of statements that may have defined V most recently before S. The closure, hence, contains all variable definitions which directly affect the variables containing time at set time API calls. We mark all statements in the closure as part of the *TICS*.

For example in code listing 3, we first mark line 13 (S_{13}), `settime(x)` as the end of a TICS pushing (S_{13},x) to C_{ST}. $DEFS(S_{13}, x)$ will contain all the definitions of x that reach line 13, *i.e.,* lines 2 and 5 and they are marked as part of the TICS.

Case 3: Time Arithmetic. Recall that timer arithmetic can be done using either fixed APIs (*e.g.,* `ktime_sub(`t_1`,`t_2`)`) or general integer arithmetic. This case is challenging because it is not obvious which two time variables are involved in that arithmetic. We make a key observation that only the code between getting the two timestamps used in the arithmetic expression forms a TICS– the arithmetic itself is not a TICS. This observation motivates our detection scheme as follows.

Algorithm 3 TICS identification for Case 3: Time arithmetic

```
Require: Program $P$, Callgraph $C$
Require: Statement-variable tuples $(S_{GT}, V_t)$ that get time
1: Time critical statements $TICS \leftarrow \emptyset$
2: for all $(S_{GT}, V_t)(S'_{GT}, V'_t); S_{GT} \neq S'_{GT}$ do
3:     if $USES^+(S_{GT}, V_t) \cap USES^+(S'_{GT}, V'_t) \neq \emptyset$ then
4:         $TICS = TICS \cup$ all statements between $S_{GT}, S'_{GT}$
5:     end if
6: end for
7: return $TICS$
```

We first build $USES(S,V)$ which is the set of statements that may use the value of V computed at $S, \forall S \in P$. Now, there are a few APIs to read the current system time, *e.g.,* `getnstimeofday` is used by the kernel to get the time, and `SystemClock.elapsedRealTime` is used by both the Android framework and apps. For each use of these APIs in a statement S_{GT} with the returned time arguments V_t, we find the transitive closure $USES^+(S_{GT}, V_t)$ of the def-use chain ($USES$ set), *i.e.,* we find all statements that are directly or indirectly flow dependent on statement S_{GT}.

If timestamps are obtained at n locations, n closures are computed, corresponding to the n timestamps read. Now, if an arithmetic statement is contained in two different closures, we know that the arithmetic statement contains, and is affected by, variables whose values are either timestamps or a function of the timestamps of the closures it is involved in. All such pairs of timestamps are marked as the start and end of a *TICS* (lines 5-11).

Note that while three or more timestamps can potentially be involved in some arithmetic (we have not seen such cases), the algorithm requires no change as pairwise set intersection will capture all statements in the critical section resulting from such multiple timestamps.

5.5 Identifying Protected Statements

When a protection mechanism is enabled, *e.g.,* a wakelock is held, or interrupts are disabled, all statements until it is disabled are protected by it. Such statements can be detected using a variation of the reaching definitions dataflow analysis as in [25]. The key idea is that enabling and disabling each mechanism can be transformed to assignments of values "1" and "0" to a special mechanism variable, which initially has a value of "0". Afterwards, the state of all protection mechanisms that reach a statement can be easily observed via the reaching definitions dataflow analysis, which determines if the statement is protected. This analysis adds all of the protected statements into the *SAFE* set, which are compared with *TICS* statements to detect SITB as shown in Figure 1.

Table 2: Summary of the analyses in KLOCK.

Analysis	LOC	Time (s)
`clang` Compilation	-	71
`llvm-link` Linking	-	651
Alias analysis	-	164
KLOCK CallGraph	552	2
KLOCK *TICS* Case 1: Callback	164	2
KLOCK *TICS* Case 2: Set time	1489	246
KLOCK *TICS* Case 3: Get time	1101	501
KLOCK *SAFE*: RD Analysis	717	11
KLOCK Other	1196	-
Total	5219	1648

5.6 Limitation and Generality

KLOCK currently does not deal with wakelocks that take timeout parameters; statically finding the end of such protected regions is difficult. KLOCK can detect protected code regions due to the first two mechanisms in §2.1, but not due to suspend callbacks via conditional variable manipulation. Detecting such cases requires involved range analysis (*e.g.,* [27]). Finally, KLOCK does not detect SITBs in timed logging. We leave these as future work.

Although we focus on the Android Linux kernel, the KLOCK design is quite general and can be applied to detect SITBs in the framework, the apps, and other software systems that are vulnerable to system suspension. For example applying the KLOCK design to analyze apps just requires building call graphs that can capture intricate callbacks that cross apps and the framework.

6 Implementation

We implemented KLOCK by adding 5 custom passes on top of the baseline LLVM compiler infrastructure version 3.3 [7], the 4 custom passes discussed in §5.4 cases 1,2, 3, and §5.5, and an additional pass for building call graph of the complete kernel, as shown in Table 2. KLOCK also runs a few standard passes such as alias analysis, control flow graph simplification and a few peephole optimizations. Before running these passes, we manually exposed the relevant APIs for bootstrapping the analyses by annotating the Linux kernel.

LLVM Passes. KLOCK is implemented in C++ in a total of 5.2 KLOC, broken down into implementing different

Table 3: **Kernels used in KLOCK evaluation.**

Phone	CPU	SoC	Version
Nexus 1	Scorpion	Qualcomm QSD8250	2.6.35.7
Nexus 7	ARM A9	Nvidia Tegra 3 T30L	3.1.10
Nexus 10	ARM A15	Samsung Exynos 5	3.4.5
Nexus S	ARM A8	Samsung Hummingbird	2.6.35.7
x86	x86_64	–	4.1

Table 4: TICS (U)sage, SITB (R)eports generated by KLOCK, and confirmed (B)ugs for each case in the Android kernels. No double-counting of same bug in different kernels.

Category in kernel	Time Callback			Time Setting			Time Aritmetic		
	U	R	B	U	R	B	U	R	B
arch	4	2	1	0	0	0	2	1	1
block	4	4	0	0	0	0	0	0	0
drivers	41	23	3	3	1	0	60	50	48
fs	10	8	0	3	1	0	3	2	0
init	0	0	0	1	1	0	0	0	0
kernel	29	15	0	2	2	0	10	8	2
mm	2	1	0	0	0	0	0	0	0
net	29	29	0	0	0	0	32	9	1
security	0	0	0	0	0	0	2	0	0
sound	4	4	0	0	0	0	6	3	3
tools	1	0	0	0	0	0	4	4	4
Total	123	86	4	9	5	0	120	78	59

passes as shown in Table 2.

7 Evaluation

Our evaluation of KLOCK answers the following questions: (1) How long does it take KLOCK to analyze a large system such as the Linux kernel? (2) Is KLOCK effective in finding sleep-induced time bugs? (3) What causes KLOCK to generate false positive reports? All experiments were conducted on an Ubuntu Linux machine with an Intel 8-Core 2.33 Ghz CPU and 16 GB memory.

Performance. The execution time of KLOCK in analyzing the Linux kernel version 3.1.10 and the time breakdown into all the phases are shown in Table 2. The results show that KLOCK can analyze a large system in a fairly reasonable amount of time.

7.1 Finding Sleep-Induced Time Bugs

Since KLOCK analyzes the entire compiled kernel at once, we apply KLOCK to 5 different kernels, *i.e.,* with different configuration options and/or kernel versions, to increase the coverage of the entire Linux kernel. Table 3 lists the five kernels used in the evaluation, for four popular phones, Nexus 1, Nexus 7, Nexus 10, Nexus S, with default configuration and an x86 kernel with `allyesconfig` that has wakelocks enabled. All four phones have ARM CPUs but have different SOCs.

Table 4 summarizes the bug finding results. Each file containing time related API is counted as one time usage instance, and is counted as an SITB bug if it contains at least one statement in the *BUG* set output by KLOCK.[2] Every number in the table shows the total number of unique instances of usages/bugs across the five kernels. The number of instances and bugs for each of the three time usages is broken down according to the top level directories in the kernel. The usage number for each usage excludes time usages using safe APIs. We observe that time callback and time arithmetic occur 123 and 120 times in the five kernels, but time setting is used rarely, only 9 times by a few kernel components.

After KLOCK generated bug reports, we manually analyzed them and marked them as either false positives or bugs. In total we found 63 bugs out of which we found 14 have been fixed and 7 files with bugs have been removed from later versions of the kernel. We reported the remaining 42 bugs to the Linux kernel mailing lists, and out of the 7 developers who have replied so far, all confirmed the corresponding bugs and accepted our patches. We now describe these 63 time bugs that KLOCK found.

Measuring pulse width. KLOCK found 6 similar bugs in remote control receiver drivers – 4 in the LIRC subsystem `drivers/staging/media/lirc/` and 2 in drivers of streamzap remote `drivers/media/rc/streamzap.c` and DHT11 temperature and humidity sensor `drivers/iio/humidity/dht11.c` which measure the width of received pulse using time arithmetic. In Listing 5, the data being received is encoded in the width of the received pulse (lines 6, 7, 14-18). If the SOC suspends or if the time is reset before line 3, the width of the pulse `deltv` calculated at line 4 will be incorrect resulting into wrong value to be saved in `rx_buf` (line 18).

Measuring clock rate. KLOCK detected 5 bugs in radio, IDE and sound drivers where the drivers calculate the input clock rate using time arithmetic. Due to SITBs, the resulting clock rate measured will be incorrect.

Measuring delay. KLOCK found 4 similar bugs in IrDA chipset drivers, `drivers/net/irda/`, where the driver measures the processing delay `diff` (line 9 in code listing 6) and compares it against minimum turnaround time `mtt`, (line 10). If `mtt` is larger than `diff`, then the frame is transmitted after (`mtt - diff`) microseconds (lines 10-12). If the time is set to a time before computing line 8, then `diff` may become negative, causing unnecessarily large delay in transmitting the frame.

The SASEM USB IR remote control driver `drivers/staging/media/lirc/lirc_sasem.c` ig-

[2]We conservatively use file as a unit in counting bugs to avoid inflating the bug count. In time arithmetic, one start time is often used in arithmetic with many other timestamps, and each could have been counted as a separate bug.

Listing 5: drivers/staging/media/lirc/lirc_sir.c: Using wallclock to measure pulse width.

```
 1 static irqreturn_t sir_interrupt(int irq, void *
       dev_id) {
 2   if (status & (UTSRO_RFS | UTSRO_RID)) {
 3     do_gettimeofday(&curr_tv);
 4     deltv = delta(&last_tv, &curr_tv);
 5     if (status&UTSRO_RID) {
 6       add_read_queue(0, deltv-n*TIME_CONST);//
            space
 7       add_read_queue(1, n*TIME_CONST); //pulse
 8       n = 0;
 9       last_tv = curr_tv;
10     }
11   }
12   return IRQ_RETVAL(IRQ_HANDLED);
13 }
14 static void add_read_queue(int flag, unsigned
       long val) {
15   int newval;
16   newval = val & PULSE_MASK;
17   ..
18   rx_buf[rx_tail] = newval;
19 }
```

Listing 6: drivers/net/irda/nsc-ircc.c: Using wallclock to measure processing delay.

```
 1 static netdev_tx_t nsc_ircc_hard_xmit_fir(struct
 2   sk_buff *skb, struct net_device *dev) {
 3   ..
 4   // Start transmit only if there is currently no
          transmit going on
 5   if (self->tx_fifo.len == 1) {
 6     mtt = irda_get_mtt(skb);
 7     if (mtt) {
 8       do_gettimeofday(&self->now);
 9       diff = (self->now.tv_sec-self->stamp.tv_sec
            )*USEC_PER_SEC + (self->now.tv_usec -
            self->stamp.tv_usec);
10       if (mtt > diff) {
11         mtt -= diff;
12         udelay(mtt);
13       }
14     }
15     // Transmit frame
16     nsc_ircc_dma_xmit(self, iobase);
17   }
18 }
```

nores an input if it arrives in 250 ms since the last input. Because of unexpected discontinuities in wall clock time, the driver may end up ignoring inputs not in the 250 ms range. A similar bug was found in input driver for SoundGraph iMON IR, `drivers/media/rc/imon.c`.

Poll and wait until timeout. Linux control and measurement device interface driver `drivers/staging/comedi/drivers/serial2002.c`, shown in Listing 7, polls serial connected hardware at line 9. If there is no new data, it takes the current timestamp `now` at line 13, calculates `elapsed` at line 14 relative to `start`, obtained at line 4, and breaks the loop if `elapsed` is larger than the `timeout` (lines 15-16). KLOCK correctly flagged time arithmetic at line 14 as a bug since before reading `now` at line 13, if the time is set to a past time stamp, the driver will get stuck spinning in the while loop much longer than the intended `timeout` (typically 1 ms, not shown in Listing 7).

Listing 7: drivers/staging/comedi/drivers/serial2002.c: Using wallclock to poll and wait until timeout.

```
 1 static void serial2002_tty_read_poll_wait(struct
       file *f, int timeout) {
 2   struct poll_wqueues table;
 3   struct timeval start, now;
 4   do_gettimeofday(&start);
 5   poll_initwait(&table);
 6   while (1) {
 7     long elapsed;
 8     int mask;
 9     mask = f->f_op->poll(f, &table.pt);
10     if (mask & (POLLRDNORM | POLLRDBAND | POLLIN
           | POLLHUP | POLLERR)) {
11       break;
12     }
13     do_gettimeofday(&now);
14     elapsed = 1000000 * (now.tv_sec - start.
           tv_sec) + now.tv_usec - start.tv_usec;
15     if (elapsed > timeout)
16       break;
17     set_current_state(TASK_INTERRUPTIBLE);
18     schedule_timeout(((timeout - elapsed)*HZ)
           /10000);
19   }
20   poll_freewait(&table);
21 }
```

The accepted patch for the bug uses `CLOCK_MONOTONIC` which ignores system suspend and cannot be reset by user setting the time or by NTP as discussed in §2.2.

Similar bugs were found in 7 other drivers.

Generating waveform. KLOCK detected both case 1 and case 3 time bugs in the Linux control and measurement device interface driver `drivers/staging/comedi/drivers/comedi_test.c` in code listing 2 which was already discussed in §4.2.1.

Msm, vibrator, and timed gpio drivers. The code snippet in Listing 8, from the msm7k serial device and console driver, is responsible for turning off the UART clock once the transmit buffer is empty. This function first verifies if the clock is on (line 4) and then sets the clock state to `MSM_CLK_REQUEST_OFF` signifying that it is requested to be turned off (line 5). It then registers a timer callback function `msm_serial_clock_off` that must be called after `clock_off_delay` seconds (line 6). This callback function verifies the state of clock to be `MSM_CLK_REQUEST_OFF` (line 12), and checks if the transmit buffer is empty (line 13). If so, the clock is disabled and its state is set to off (lines 14 and 15), otherwise the callback function is rescheduled to get called again in `clk_off_delay` seconds (line 17).

SITB occurs if the CPU suspends before the timer fires and the callback function is executed. In that case, even if the transmit buffer is empty, the UART clock would unnecessarily remain turned on.

The Android kernel exposes a special timer API `android_alarm` which uses high-resolution timer to trigger an event when the CPU is active and additionally also sets an RTC wakeup alarm when the CPU is about to suspend. Switching to Android timer API `android_alarm_init` from `hrtimer_start` (at line 6)

Listing 8: drivers/serial/msm_serial.c: Unprotected use of timer callback wastes energy.

```
//request turning off clock once TX is flushed
void msm_serial_clock_request_off(struct
      uart_port *port) {
  clk_off_timer.function = msm_serial_clock_off;
  if (msm_port->clk_state == MSM_CLK_ON) {
    msm_port->clk_state = MSM_CLK_REQUEST_OFF;
    hrtimer_start(clk_off_timer,clk_off_delay,
         HRTIMER_MODE_REL);
  }
}
//clock off if TX buffer is empty, else
      reschedule
static enum hrtimer_restart msm_serial_clock_off(
      struct hrtimer *timer) {
  int ret = HRTIMER_NORESTART;
  if (msm_port->clk_state==MSM_CLK_REQUEST_OFF) {
    if (uart_circ_empty(xmit)) {
      clk_disable(msm_port->clk);
      msm_port->clk_state = MSM_CLK_OFF;
    } else { //reschedule
      hrtimer_forward_now(timer, clk_off_delay);
      ret = HRTIMER_RESTART;
    }
  }
  return HRTIMER_NORESTART;
}
```

fixes the SITB since the CPU will be woken up just in time to turn off the UART clock.

Similar bugs were found in vibrator driver `arch/arm/mach-msm/msm_vibrator.c` and timed gpio driver `drivers/staging/android/timed_gpio.c`.

Leaky bucket. The driver for Beeceem WIMAX chipset used by Sprint 4G, `drivers/staging/bcm/LeakyBucket.c`, implements a routine related to the Leaky Bucket algorithm. As shown in the code snippet in Listing 9, function `UpdateTokenCount()` controls the number of packets that can be transmitted in a fixed time period. Line 3 reads the current time in `tv` and line 4 computes the number of seconds passed since the token count was last updated and stores it in `currentTime`. If `currentTime` is non-zero, the current token count is incremented by the number of packets that can be transmitted in `currentTime`, and the last update time is set to current time (line 7).

If the token accounting semantics is to include CPU sleep time, then if the CPU sleeps after line 5, line 6 will be executed after the CPU wakes up and under-calculates the tokens accumulated. If the token accounting semantics is to exclude the CPU sleep time, then the token accounting is correct in the current invocation of the function if the CPU does not sleep before line 3. But if the CPU sleeps after line 3, in the next invocation of function `UpdateTokenCount()`, `currentTime` calculation (line 4) would include the sleep time, again resulting in incorrect token calculation.

Benchmarking and stats reporting. KLOCK detected the sleep bug in the the memcpy benchmark discussed in §3. Similar bugs were detected in 29 other places.

Listing 9: drivers/staging/bcm/LeakyBucket.c: Incorrect token accounting due to SITB

```
 1 //Called every time before transmitting packets.
 2 static void UpdateTokenCount() {
 3   do_gettimeofday(&tv);
 4   currTime = tv.tv_sec-pcktInfo.lastUpdate.tv_sec
        ;
 5   if(currTime!=0) {
 6     pcktInfo.tokens+= pcktInfo.maxRate*currTime;
 7     memcpy(pcktInfo.lastUpdate,&tv,sizeof(struct
          timeval));
 8     if(pcktInfo.tokens>=pcktInfo.maxBucketSize)
 9       pcktInfo.tokens=pcktInfo.maxBucketSize;
10   }
11 }
```

Miscellaneous. 2 other time arithmetic bugs were found in infiniband driver `drivers/infiniband/hw/mlx4/alias_GUID.c`, and storage controller driver `drivers/scsi/3w-9xxx.c`. We skip their details due to page limit.

7.2 False Positives

KLOCK reported 106 time manipulation instances to contain SITBs, which upon manual analysis, turned out to be false positives. We note the false positive rate of 63% is a reasonable tradeoff for the high coverage of static analysis (*e.g.*, [27] reports finding 11 bugs out of 741 reports, [20] reports finding 252 bugs out of 955 reports). We found three reasons that cause KLOCK to generate false error reports:

System suspension does not affect program semantics. We found false positives in cases where the program semantics are not impacted by system suspension during time manipulation. Marvell wireless LAN device driver `drivers/net/wireless/mwifiex/wmm.c`, for example, just calculates a random number by performing time arithmetic. Similarly, the kernel process scheduler registers a timer callback for scheduling a new process at the end of a time slice, but even if the CPU suspends in the middle, the delay in callback execution will have no impact on the scheduler semantics. Reducing such false positives requires understanding program semantics.

System calls. KLOCK flags system calls such as `sys_settime`, `sys_utimes` as bugs. This is because KLOCK only analyzes the entire Linux kernel, and these system calls are effectively wrappers to the actual time setting APIs and are meant to be invoked by user-space programs; by themselves they do not enable any suspension prevention mechanism. Such system call usages can cause sleep-induced time bugs in the user-space programs calling them if they do not use proper suspension prevention mechanisms.

Dependence on system suspension code. Requesting firmware `drivers/base/firmware_class.c`, for example, holds a semaphore shared with the code that disables usermodehelper which lies on the suspension code path.

Reducing such false positives requires tracking the state of all global conditional variables and semaphores shared with all the code on suspension code path. We leave it as future work.

7.3 The Significance of SITBs

SITBs occur in all software layers in the mobile ecosystem. They can impact both performance and program correctness. In particular, out of the 63 bugs KLOCK found, 30 are benchmarking bugs, and the remaining 33 bugs either impact performance (including energy) or correctness of device drivers.

Correctness related. (i) 6 drivers under "Measuring pulse width" decode a received signal by measuring the width of a pulse. SITBs make them measure the width incorrectly, hence reading the received data incorrectly. (ii) 2 drivers under "Measuring delay" incorrectly ignore user input. (iii) 5 drivers under "Measuring clock rate" measure the clock rate incorrectly. These are mostly radio drivers needed to detect the incoming clock rate to decode data. The data decoded will be wrong if the measured clock rate is incorrect.

Performance related. (i) 8 drivers under "Poll and wait until timeout" category cause the driver to spin for a long time, making the device unusable. (ii) 4 drivers under the "Measuring delay" category cause the driver to sleep for a long time, making the device unusable. (iii) 3 drivers, msm, vibrator and timed_gpio, keep the device on longer than necessary, wasting energy.

In summary, none of these bugs crash the kernel, but they are serious bugs affecting the correctness or performance of the kernel.

8 Related Work

Hunting bugs in Linux is a topic almost as old as Linux itself [23]. Sleep-related bugs (in Linux) on smartphones is a relatively new and exciting area. Previous work has focused on sleep bugs that result in energy leaks, or energy bugs. Pathak *et al.* were the first to discuss the significance of energy bugs in smartphones [24] and developed a taxonomy of smartphone energy bugs. In [25], Pathak *et al.* studied no-sleep energy bugs, a class of sleep bugs caused by not releasing wakelocks in apps which causes SOC/CPU to stay awake, and developed a detection tool based on reaching definitions dataflow analysis. In [16], Jindal *et al.* studied sleep conflicts, a class of sleep bugs in device driver code that cause phone devices to stay in an active power state till indefinite due to unexpected SOC/CPU suspension, and proposed a system to perform runtime avoidance of sleep conflict. In [17], Jindal *et al.* developed a taxonomy of sleep disorder bugs, which includes no-sleep, over-sleep and under-sleep bugs. SITBs are first class of over-sleep bugs studied.

Carat [22] treats apps as blackboxes and performs collaborative debugging to identify "energy hog" apps based on observed behavior of an app running on many phones.

In contrast to these previous work, we study a new class of sleep-related bugs, sleep-induced time bugs, that manifest as logical errors and alter the intended program behavior.

Our work relies in part on finding ordering relationships between actions on time-related system calls, variables that are a function of time values, and wakelock acquires and releases. Engler's MC language [11] builds upon the Metal state-machine language and allows writing compiler extensions for static checking of temporal relationships between program actions. As described in §5.4, Algorithms 2 and 3, our techniques require finding transitive closures of use-def chains and, in Algorithm 3, additionally finding the pair-wise intersections of the closures which are beyond statically defined state machines created when compiling MC checks.

We note that [26] and [18], among others, perform static race detection, and in the course of doing this identify the sets of locks held at a location. While it might be possible to take their analysis and adapt it to our needs, we find our simple data flow based algorithm to be sufficient and efficient.

9 Conclusion

This paper presents the first study of a new class of sleep-related bugs on smartphones, sleep-induced time bugs, that can occur in all layers of smartphone software, *i.e.,* the kernel, framework, and apps. A SITB happens when the phone is suspended in the middle of a time critical section that manipulates time and as a result alters the intended program behavior. We characterize the pervasive usage of time usage in smartphone software layers, classify them into four usage patterns, and show their vulnerability to SITBs. We present the design and implementation of KLOCK, a tool that detects SITBs in large systems. KLOCK has aided in finding 63 SITBs in the Linux kernel. We have released KLOCK at `http://github.com/klock-android` for use by smartphone OS developers to test for sleep-induced time bugs.

Acknowledgment. We thank the anonymous reviewers and our shepherd Anthony D. Joseph for their constructive comments which helped to improve this paper. This work was supported in part by NSF grant CCF-1320764 and by Intel.

References

[1] Android time api. `http://developer.android.com/reference/android/text/format/Time.html,http://developer.android.com/reference/android/os/SystemClock.html`.

[2] Android timer api. `http://developer.android.com/reference/java/util/Timer.html`.

[3] Decompiling apps. `http://siis.cse.psu.edu/ded/`.

[4] Kernel manipulating time. `http://linux.die.net/man/3/timersub,http://lxr.free-electrons.com/source/include/linux/ktime.h`.

[5] Kernel querying time. `http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/include/linux/time.h`.

[6] Kernel timer api. `http://www.ibm.com/developerworks/library/l-timers-list/`, `https://lwn.net/Articles/429925/`.

[7] Llvm compiler infrastructure. `https://llvm.org`.

[8] [patch] iio: dht11: Use boottime. `http://www.spinics.net/lists/linux-iio/msg22706.html`.

[9] Speedtest.net. `https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest`.

[10] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of OSDI*. USENIX Association, 2000.

[12] D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of ACM SOSP*, 2001.

[13] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *SOSP*, 2003.

[14] Dawson Engler and Madanlal Musuvath. Model-checking large network protocol implementations. In *Proc. of USENIX NSDI*, 2004.

[15] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. of VMCAI*, 2004.

[16] Abhilash Jindal, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. Hypnos: Understanding and Treating Sleep Conflicts in Smartphones. In *Proc. of EuroSys*, 2013.

[17] Abhilash Jindal, Abhinav Pathak, Y Charlie Hu, and Samuel Midkiff. On death, taxes, and sleep disorder bugs in smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, page 1. ACM, 2013.

[18] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 13–22. ACM, 2009.

[19] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proc. of USENIX OSDI*, 2006.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006.

[21] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proc. of USENIX OSDI*, 2002.

[22] Adam J. Oliner, Anand Iyer, Eemil Lagerspetz, Sasu Tarkoma, and Ion Stoica. Collaborative energy debugging for mobile devices. In *Proc. of USENIX HotDep*, 2012.

[23] Nicolas Palix, Gaēl Thomas, Suman Saha, Christophe Calves, Gilles Muller, and Julia Lawall. Faults in linux 2.6. *ACM Transactions on Computer Systems (TOCS)*, 32(2):4, 2014.

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

[25] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of Mobisys*, 2012.

[26] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):3, 2011.

[27] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *Proc. of USENIX OSDI*, 2012.

[28] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proc. of USENIX OSDI*, 2004.