Detecting No-Sleep Energy Bugs Using Reference Counted Variables

Scott Hall
Oakland University
Rochester, Michigan, USA
wshall@oakland.edu

Suman Nataraj
Oakland University
Rochester, Michigan, USA
snataraj@oakland.edu

Dae-Kyoo Kim
Oakland University
Rochester, Michigan, USA
kim2@oakland.edu

ABSTRACT

Mobile devices have become pervasive in today's society. The range of their use has been constantly increasing, which requires more computing capability. As the computing capability of mobile devices grow, so does the need for effective power management. There has been some work on reducing the power consumption of mobile applications by detecting energy bugs. In this work, we address no-sleep energy bugs with respect to semaphore wakelocks in consideration of race conditions with synchronization using reaching definitions and parallel flow graphs. We demonstrate the approach through a case example.

KEYWORDS

Energy bugs, mobile computing, reaching definition, race condition, wakeLock

ACM Reference Format:
Scott Hall, Suman Nataraj, and Dae-Kyoo Kim. 2018. Detecting No-Sleep Energy Bugs Using Reference Counted Variables. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3197231.3197257

1 INTRODUCTION

Mobile devices have become ubiquitous in today's society. They are used for not only communication, but also many other purposes such as entertainment and web surfing. As the capacity of mobile devices grows, so does the need for effective power management. Thus, power management is critical for mobile devices. However, the longevity of battery power is limited and the energy demand well outstrips the advance of battery longevity [3].

There has been some work (e.g., [3, 4]) on reducing energy consumption in mobile applications. These areas include low power sleep states, screen modifications, optimizations of network components, and offloading computing intensive functions. The operating system of mobile devices is designed to be energy efficient by putting components to sleep as soon as they become idle, while allowing individual applications to manage components (e.g., screens, CPUs) to be kept awake. In particular, Android's WakeLock API allows developers to manage the energy consumption of components. This may lead to a mismanaging practice where a component is kept awake indefinitely, which introduces an energy bug. Such a bug is particularly hard to detect since they do not lead to crashes or malfunctions other than reducing the battery life.

A common type of energy bug is energy draining by wakelocks which keep up a thread/process awake. Wakelock bugs drain about 50-60% of the battery without a user interaction or performing any activities [6]. The work by Pathak *et al.* [6] uses the reaching definitions (RD) dataflow analysis [2] to detect no-sleep bugs (which is also known as ebugs). They use the traditional RD and consider synchronization. However, their work does not address race condition. Based on their work, we address no-sleep bugs in consideration of race condition in synchronization. Also, instead of using the traditional RD, we use reference counted RD which reduces false positives.

The remainder of the paper is structured as follows. Section 2 gives an overview of related works. Section 4 describes detecting no-sleep bugs that contain a race condition and a way to incorporate solutions with these PFGs and wakelocks that use counters and act as semaphores.

2 RELATED WORK

There is some work on detecting energy bugs. The work by Pathak et al. [6] proposes a technique to detect no-sleep bugs in smartphone applications using the class RD analysis. They describe different types of no-sleep bugs with their characteristics. We extend their work by considering synchronized race conditions and reference counted wakelocks.

The work by Agarwal et al. [1] proposed a system called MobiBug for diagnosing mobile applications for failures. The system is developed to monitor dependencies between failures and resources based on spatial spreading, statistical inference, and adaptive sampling. However, they do not address energy bugs which are different from traditional bugs in that they do not lead to an application crash, but continually drain the battery [5].

Grunwald and Srinivasan [2] present a RD analysis for programs with explicit parallel constructs (e.g., cobegin, coend) using parallel flow graphs (PFGs) representing multiple threads along with synchronization. They also consider synchronization operations (e.g., post, wait). Their work focuses on optimizing parallel programs to reduce the complexity of the RD problem through memory consistency model. We adopt their work for static analysis of RD to detect no-sleep bugs in synchronized race conditions with reference counted variables.

3 BACKGROUND

In this section, we give a background on no-sleep energy bugs and parallel flow graphs which are used in this work to detect no-sleep bugs.

3.1 No-Sleep Energy Bugs

No-sleep bugs are mainly attributed to mismanaging practices of wakelocks which act as semaphores. No-sleep bugs can be introduced by various causes. A common cause is code paths that do not release a wakelock properly because of improper execution of the path or unexpected deviation from the intended path by exception. Also, the deletion of an object does not automatically release its associated wakelocks unless specifically tasked to do so, which is often misunderstood. Exiting an application is another cause of a no-sleep bug. When an application is exited, it is not shut down, but rather still running background. It is only shut down when the task manager does so or there is an emergency system event such as low RAM or power. Sleep dilation is another cause. An application eventually releases its associated wakelocks, but only after a prolonged time. This is often caused by the event-driven nature of the mobile phone application, which is difficult to debug.

No-sleep bugs are also often caused by race conditions which are the focus in this work. A race condition occurs when multiple threads try to change shared data at the same time (i.e., threads are racing to access the data). However, the order in which the threads attempt to access the shared data cannot be expected due to thread scheduling algorithms which can swap between threads at any time. Therefore, the result of the change in data is dependent on the thread scheduling algorithm. Problems often occur when one thread does a "check-then-act" to a value and another thread changes the value in between the "check" and the "act", which makes it difficult to expect the result of the value. A typical practice to prevent this is to use a lock to ensure only one thread can access the data at a time. Other solutions are to use synchronization or prioritization. Race conditions often introduce no-sleep energy bugs. A wakelock can be shared by multiple threads racing each other to acquire it and its improper release causes a no-sleep bug.

Listing 1 shows an example which involves two threads – main thread and weather thread. The main thread is awakened by acquiring the wakelock w1 and runs the weather thread to check the weather. There was an attempt made in the code for efficient handling of the wakelock by first acquiring it in the main thread and then releasing it at the end of the main thread. The problem is that the weather thread also has access to the wakelock, which creates a race condition. It wakes up every 10 seconds while the checkWeather variable is set to true. However, in lines 16-18, it releases the wakelock before sleeping and acquires back upon waking up. However, if checkWeather has been set to false by line 7 after acquiring the wakelock in line 18 and the weather thread has been stopped by line 8, then the wakelock acquired from line 18 will never be released, which introduces a no-sleep bug. This is not a problem if the execution of the weather thread runs before line 9 because the wakelock is released line 9. However, that cannot be guaranteed by thread scheduling algorithms.

Listing 1: An Banking Application in Android

```
1    public void mainThread(){
2
3            w1.acquire();
4            checkWeather = true;
5            weatherThread.run();
6            // other code
7            checkWeather = false;
8            weatherThread.stop();
9            w1.release();
10
11       }
12
13   public void weatherThread(){
14           while (checkWeather){
15                   checkWeather();
16                   w1.release();
17                   sleep(10000);
18                   w1.acquire();
19           }
20   }
```

3.2 Parallel Flow Graphs

We use an RD analysis to detect no-sleep bugs in multi-threaded applications with synchronization. RD analysis is a method of determining a path between two nodes in which a variable is declared in the beginning node and then it is not defined again on the path until the ending node. A variable v is said to reach a point p if the path from the point immediately following v to p contains no other definitions of v. The set of variables that reach the point p is said to be the RD for p.

In order to use RD, we represent a program in parallel flow graphs (PFGs) which is a type of control flow graph (CFG) that deals with parallel computing programs using multiple threads. PFGs involve three kinds of edges – (i) sequential control edges which represent sequential processing, (ii) parallel flow edges which represent either a fork or join statement, and (iii) synchronization edges which represent paths to synchronized methods or blocks. We assume that the data exchange happens only between threads through synchronization.

Figure 1 shows an example of a PFG. In the diagram, a cobegin statement represents the beginning of a thread, while the coend statement represents the logical conclusion of it. The two dotted lines from the cobegin represent two logical threads. The dotted line from the cobegin node to node 2 can be considered as the main thread, while the other dotted line from the cobegin node to node 3 can be considered as a worker thread. Both threads can continue on to node 4 and node 5 respectively where they both eventually join to continue non-threaded processing at node 6. Note that the dotted line from node 2 to node 4 represents synchronization, meaning that one thread can exchange data with another. This causes certain problems for the RD. Let us say that the variable z is defined in node 1 and redefined in both node 2 and node 3. Then, a question is what is the value of z in node 4 as z has been killed in node 2 and 3 and thus, it is not of the reaching definitions of node 4. By the synchronization between node 2 and 4, node 2 must wait for the signal of node 4 before the values are exchanged.

In the traditional RD, a wakelock wi that is given an acquire is defined as wi = 1 and that is given a release is defined as wi = 0. RD is checked for 0 and 1 for each wakelock at the end of the code block and wakelocks that have a definition of 1 are considered to be no-sleep bugs. We extend the traditional RD by considering reference counts to reduce false positives of no-sleep bugs. The extension defines the wakelock wi as wi = wi + 1 for an acquire and wi = wi - 1 for a release. Then, RD is checked for zero and non-zero for

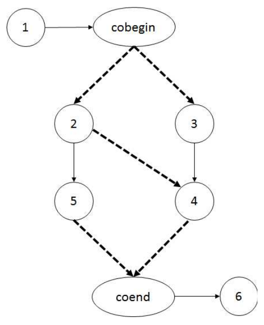

Figure 1: PFG Example

each wakelock instead of 0 and 1. Any wakelock with a non-zero definition at the end of convergence in the PFG is a possible no-sleep bug.

4 DETECTING NO-SLEEP BUGS IN RACE CONDITIONS

In this section, we describe the RD analysis with reference count to detect no-sleep bugs through a running example. Listing 2 shows the running example of a banking application. The example is developed to demonstrate a no-sleep bug introduced by improper management of wakelocks and race condition in synchronization, which can be often found in Android development.

In the code, the PowerManager class is responsible for user controlled power management through the Wakelock class. The Wakelock class can be instantiated with four different options, each concerning a specific component (e.g., CPU, screen) and energy level (e.g., screen brightness, keyboard backlight). There are two wakelocks, one partial and one full, declared at the beginning of the function. Wakelocks are activated and deactivated through the acquire() and release() methods. They are not necessarily mutually exclusive and multiple wakelocks can be held for a single component by different applications. The component may go to sleep only after all its associated wakelocks are released. Each wakelock corresponds to a user initiated event. There are two threads running in the function – the main thread and a worker thread. The main thread is responsible for executing the function, while the worker thread handles updating account information and checking for account messages. The behaviors of the worker thread run under a while loop and which behavior to execute is specified in the if-else clauses. To demonstrate a no-sleep bug, the while loop is designed to run at least two times with the else clause to be the first running. The *checkAccountMessages*() function in the else clause assumes a partial wakelock having been acquired. The partial wakelock is released after the function call of *checkAccountMessages*() and the thread is put on sleep for six seconds. It wakes up after six seconds and acquire the partial wakelock. If the value of *uai* is set to true by the main thread at line 40, the else clause in the worker thread will never be executed, and thus the wakelock of the worker thread is not released, which introduces a no-sleep bug.

The if clause in the worker thread introduces another no-sleep bug with respect to the synchronization in the main thread, which is more difficult to discover. The if clause involves the *wait*() function for the worker thread to wait until the main thread is synchronized. This is based on the assumption that the worker thread runs at a faster pace than the main thread. However, it is very difficult to predict thread scheduling. In fact, the worker thread may not run until the main thread reaches the release statement at line 45 after the synchronization. In this case, the if clause in the worker thread will be never executed and the full wakelock acquired by the main thread during the synchronization will never be released, which introduces another no-sleep bug.

Listing 2: An Banking Application in Android

```
1 public void maintainBankAccount(){
2
3   PowerManager powerManager = (PowerManager) getSystemService(POWER_SERVICE);
4   Wakelock pw = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK);
5   Wakelock fw = powerManager.newWakeLock(PowerManager.FULL_WAKE_LOCK);
6   boolean kr = true;
7   boolean uai = false;
8
9   new Thread(new Runnable() {
10      public void run() {
11
12      while (kr){
13
14        if (uai){
15            wait();
16            updateAccountInfo();
17            fw.release();
18        }else{
19            checkAccountMessages();
20            pw.release();
21            sleep(6000);
22            pw.acquire();
23
24            if (uai)
25                uai = false;
26            else
27                uai = true;
28        }
29      }
30    }
31  }).start();
32
33    BankAccountData bad = loadInitialBankData();
34    fw.acquire();
35
36    doEventDrivenWork();
37    doMoreEventDrivenWork();
38    doEvenMoreWork();
39
40    uai = true;
41    synchronize(this){
42       fw.acquire();
43       updateAccountInfo();
44    }
45    pw.release();
46    kr = false;
47}
```

As demonstrated, there are no-sleep bugs present in the code, one concerning the partial wakelock and one concerning the full wakelock. The traditional RD can easily detect the no-sleep bug associated with the partial wakelock. However, the same is not true for the bug associated with the full wakelock, which cannot be detected until it has reached the convergence where the status of the full wakelock no longer changes. To detect the bug, we present two sets of PFG analysis, one without reference count and one with reference count.

Figure 2 shows the PFG without reference count. In the graph, nodes 2 and 11 to 13 represent the main thread where node 2 corresponds to line 3 and nodes 11 and 12 specify to line 34 and 42 respectively where the full wakelock is acquired. Similarly, nodes 3 to 10 represent the worker thread where node 4 corresponds to line 14, node 5 represents to line 15 where the wait function is declared, node 6 corresponds to line 17 where the full wakelock is released, node 7 specifies line 20 where the partial wakelock is released, node 8 corresponds to line 21, node 9 represents line 22 where the partial wakelock is acquired, and node 10 represents the end of the while loop in line 29. Finally, node 14 specifies the co-end node which is the termination point for both threads. The dashed line between nodes 12 and 5 represents the synchronization in line 41. We are most concerned with the Out set of variables in node 14 (i.e., Out(14)) which represents the set of the definitions that reach the end of node. In the figure, Out(14) is defined as {fw6, pw9} which means that the node has RD of the full wakelock with a release definition and the partial wakelock with an acquire definition. The partial wakelock with an acquire definition suggests that there is highly likely a no-sleep bug, which requires a further examination of the code containing the partial wakelock.

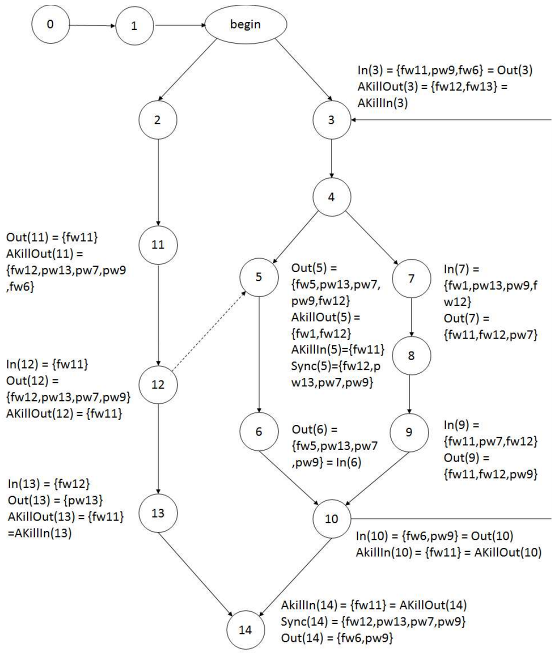
0
1
begin
2
3
In(3) = {fw11,pw9,fw6} = Out(3)
AKillOut(3) = {fw12,fw13} = AKillIn(3)
4
11
Out(11) = {fw11}
AKillOut(11) = {fw12,pw13,pw7,pw9,fw6}
5
Out(5) = {fw5,pw13,pw7,pw9,fw12}
AkillOut(5) = {fw1,fw12}
AKillIn(5)={fw11}
Sync(5)={fw12,pw13,pw7,pw9}
7
In(7) = {fw1,pw13,pw9,fw12}
Out(7) = {fw11,fw12,pw7}
8
12
In(12) = {fw11}
Out(12) = {fw12,pw13,pw7,pw9}
AKillOut(12) = {fw11}
6
Out(6) = {fw5,pw13,pw7,pw9} = In(6)
9
In(9) = {fw11,pw7,fw12}
Out(9) = {fw11,fw12,pw9}
13
In(13) = {fw12}
Out(13) = {pw13}
AKillOut(13) = {fw11} =AKillIn(13)
10
In(10) = {fw6,pw9} = Out(10)
AkillIn(10) = {fw11} = AKillOut(10)
14
AkillIn(14) = {fw11} = AKillOut(14)
Sync(14) = {fw12,pw13,pw7,pw9}
Out(14) = {fw6,pw9}

Figure 2: PFG without Reference Count

Figure 3 shows the PFG with reference count. Note that in this analysis, the definitions for wakelocks may have a value greater than 1. In the figure, Out(14) is defined as {fw6, pw9, fw11}. fw6 and pw9 are the same as the non-reference count analysis in Figure 2. fw11, which represents the full wakelock acquired in node 11 (line 34), is added as the full wakelock has been acquired twice (also acquired in node 12 (line 42), but released only once in node 6 (line 17). This definition has a value of two. Since the full wakelock has been released only once, there is a high likelihood of having a no-sleep bug with respect to the full wakelock, which is not detected in the non-reference count analysis above.

In(n) and Out(n) definitions are important for leading to the end value. However, the primary components for a synchronized RD solution are AKillOut(n) and AKillI(n) definitions along with Synch(n). AKillIn(n) defines the set of definitions of the variables that are always killed. These variables are propagated by synchronization edges and later killed in node *n*. Similarly, AKillOut(n) defines the set of definitions of the variables that are always killed and the variables are propagated by synchronization edges, but unlike AKillIn(n), they are killed at the end of node *n*. Synch(n) specifies the set of the synchronized definitions that must reach the node *n* via Pred_sy which defines the set of the core functional nodes that may post to a wait node. AKillOut(n) and AKillI(n) definitions are used to track accumulated kills which are important in a loop environment. Real-world scenarios often contain many loops along with synchronization which greatly increases the complexity.

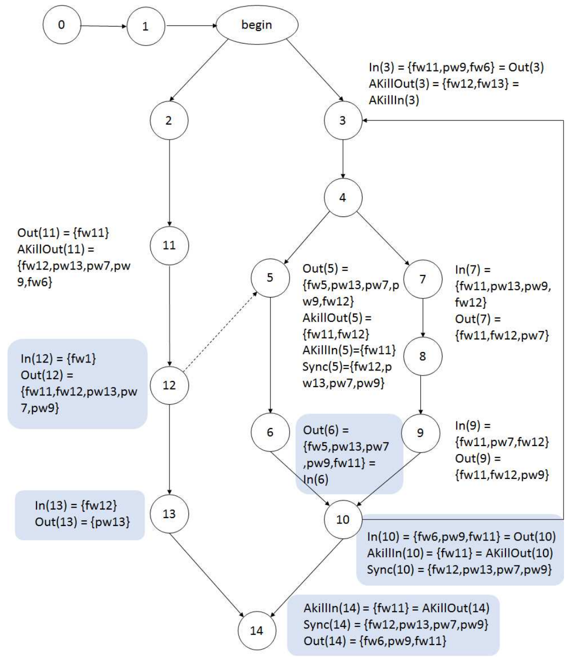
0
1
begin
2
3
In(3) = {fw11,pw9,fw6} = Out(3)
AKillOut(3) = {fw12,fw13} = AKillIn(3)
4
11
Out(11) = {fw11}
AKillOut(11) = {fw12,pw13,pw7,pw9,fw6}
5
Out(5) = {fw5,pw13,pw7,pw9,fw12}
AkillOut(5) = {fw11,fw12}
AKillIn(5)={fw11}
Sync(5)={fw12,pw13,pw7,pw9}
7
In(7) = {fw11,pw13,pw9,fw12}
Out(7) = {fw11,fw12,pw7}
8
12
In(12) = {fw1}
Out(12) = {fw11,fw12,pw13,pw7,pw9}
6
Out(6) = {fw5,pw13,pw7,pw9,fw11} = In(6)
9
In(9) = {fw11,pw7,fw12}
Out(9) = {fw11,fw12,pw9}
13
In(13) = {fw12}
Out(13) = {pw13}
10
In(10) = {fw6,pw9,fw11} = Out(10)
AkillIn(10) = {fw11} = AKillOut(10)
Sync(10) = {fw12,pw13,pw7,pw9}
14
AkillIn(14) = {fw11} = AKillOut(14)
Sync(14) = {fw12,pw13,pw7,pw9}
Out(14) = {fw6,pw9,fw11}

Figure 3: PFG with Reference Count

5 CONCLUSION

We have presented an approach for detecting no-sleep energy bugs in synchronized race condition using an RD analysis. We have also shown that use of reference counted variables can detect potential no-sleep bugs that cannot be detected by the non-reference count analysis. We plan to expand experiments with real-world examples which involve more synchronization with complex threading of wakelocks. We also plan to investigate try-catch statements which is a primary source of trouble for developers as the code takes unexpectedly a route that was not anticipated [6]. This is an area that goes hand-in-hand with the problem of race conditions and serves to greatly increase the complexity. Time variables become important as an increasing number of user applications interact with power-conscious operating systems, which we feel that the majority of future energy savings would come from this area. We shall look into considering them in RD.

REFERENCES

[1] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. 2010. There's an app for that, but it doesn't work. Diagnosing Mobile Applications in the Wild. In *Proceedings of the 9th ACM Workshop on Hot Topics in Networks*. Monterey, CA, 1–6.

[2] D. Grunwald and H. Srinivasan. 1993. Data Flow Equations for Explicitly Parallel Programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, 159–168.

[3] P. Ljung. 2011. Opportunities for Energy Savings in Mobile Devices. In *Proceedings of the 22nd IEEE International Symposium on Personal Indoor and Mobile Radio Communications*. Toronto, Canada, 2394–2401.

[4] A. Min, R. Wang, J. Tsai, M. Ergin, and T. Tai. 2012. Improving Energy Efficiency for Mobile Platforms by Exploiting Low-Power Sleep States. In *Proceedings of the 9th conference on Computing Frontiers*. New York, NY, 133–142.

[5] A. Pathak, Y. C. Hu, and M. Zhang. 2011. Bootstrapping Energy Debugging for Smartphones: A First Look at Energy Bugs in Mobile Devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. Cambridge, MA, 1–6.

[6] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. 2012. What is Keeping My Phone Awake?: Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. New York, NY, 267–280.