WakeScope: Runtime WakeLock Anomaly Management Scheme for Android Platform

Kwanghwan Kim, Hojung Cha

Department of Computer Science
Yonsei University
Seoul, Korea

{kwanghwan, hjcha}@cs.yonsei.ac.kr

ABSTRACT

Android provides a WakeLock mechanism for application developers to ensure the proper execution of applications without having to enter the sleep state of a device. When using the WakeLock mechanism, application developers should bear the responsibility of adequately releasing the acquired lock. Otherwise, the energy will unnecessarily be wasted due to a locked application. This paper presents a scheme, called WakeScope, to handle WakeLock misuse. The scheme is designed to detect and notify of a misuse case of WakeLock handling, which may arise with an application and even with an Android runtime system, and thus provides a practical tool to prevent energy waste in mobile devices. Our experiments with real applications show that WakeScope accurately detects the misused case, with runtime overhead of approximately 1.2% in CPU usage.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools (e.g., data generators, coverage testing)*

General Terms

Management, Detection

Keywords

Smartphones, Mobile, Energy, WakeLock, Android

1. INTRODUCTION

Despite the rapid development in smartphone technology, efficient energy management is required to prolong limited battery lifetime. As part of a system-level approach for battery management, Android, for instance, employs an aggressive sleeping mechanism to minimize the use of battery resources. While the device is in use, Android ensures that all the components work properly with full functionality. Otherwise, Android maintains the system in a sleep state by forcing all the components, including CPU, to idle state; thus, minimizing the power consumption. This policy should, however, be used with care. For example, if the device enters the sleep state in the middle of using network connections, the application will not work properly since all the components become idle. The WakeLock mechanism [1] in Android is designed to prevent this kind of application scenario.

WakeLock is a mechanism that guarantees a mobile device wakes up without entering into the sleep state when the application is running [2]. The WakeLock mechanism in Android is provided in the form of APIs. The WakeLocked portion of the application is guarded by the lock acquisition and release. The application developer is responsible for the correct usage of the primitives; otherwise, the energy will be drained unnecessarily. The WakeLock is a necessary mechanism for system operation, but ensuring its right usage may be a significant burden to an application developer.

If an acquired WakeLock is not released adequately, energy waste occurs since the device cannot enter the sleep state. The battery drainage due to misuse of WakeLock is reported to be at least 5% to 25% per hour [3]. Even when the acquired WakeLock is explicitly released, there are situations in which the application may fall into an unexpected situation, such as an exception, where the acquired WakeLock would not be released. To prevent this, the application developer should consider all the possible cases of exceptions while writing the code, which is certainly not a trivial task. Thus, an effective scheme to detect and manage the mishandling of WakeLock would be very helpful for an application developer.

Many attempts to manage WakeLock mishandling have been made applying various approaches [3, 4]. The previous work detects WakeLock misuse by analyzing the application source at compile time and informs the developer of the causes of the problem. Although this approach may solve the problem from the developer point of view, there is nothing users could do while using the application. Users must endure the energy waste, due to the misuse of the WakeLock mechanism, until the problem is fixed by the developer. This implies that the problem must be detected at runtime and dealt with appropriately by the user.

In this paper, we present a runtime scheme, called WakeScope, to manage the WakeLock mishandling problem. WakeScope detects the misused case of WakeLock generated from the application and also from the Android system in runtime, and notifies a user of the detection. For this, WakeScope continually tracks the behavior of WakeLock acquisition and release. The acquisition of WakeLock means that there is a critical job that must be executed without entering into the sleep state. With the acquisition of WakeLock, if the application in running state stops without releasing the WakeLock, we consider that the application has a

Table 1. WakeLock Types in Android

WakeLock Type	CPU	Screen	Keyboard
PARTIAL	On	Off	Off
SCREEN_DIM	On	Dim	Off
SCREEN_BRIGHT	On	Bright	Off
FULL	On	Bright	Bright

WakeLock mishandling problem. In this study, we define this phenomenon as "WakeLock anomaly" and propose a runtime scheme to detect and handle it.

The contributions of our work are as follows:

- We propose a method that can accurately track the WakeLock behavior used by both the application and Android system in runtime.
- Contrary to prior work that detects the WakeLock mishandling in compile time, we propose a method that can detect the problem in runtime.
- By detecting and adequately handling the problem, we provide a practical solution to prevent the energy waste caused by the WakeLock mishandling.

The paper is structured as follows. The Background is presented in Section 2. Section 3 provides an overview of WakeScope. WakeLock behavior tracking and WakeLock anomaly detection are discussed in Sections 4 and 5, respectively. Section 6 presents the solution for the WakeLock anomaly. The system is evaluated in Section 7. Section 8 discusses the related work, and Section 9 concludes the paper.

2. BACKGROUND

2.1 Android Power Management

Android power management is built on top of Linux power management. Linux basically manages the energy consumption of the device with the suspend state, in which all components of the device are maintained idle so that minimum power is consumed. The original Linux power management is, however, not suitable for a mobile device that has limited battery capacity. This means that the suspend state should be more aggressively applied to prolong the battery life time in the mobile device.

Android power management is indeed more aggressive than Linux power management. If there is no interaction between users and the device, the device changes to the sleep state and the system transits to the suspend state. If user interaction, such as a power key pressed or keyboard touch, occurs in this state, the device is put into the awake state and the system changes to the resume state, upon which all the device components then work properly.

2.2 Android WakeLock Mechanism

Table 1 shows four types of WakeLocks and the components that are controlled by each WakeLock type [2]. The components are CPU, screen light and keyboard backlight. All the WakeLocks keep the CPU always on, and the WakeLocks except for PARTIAL_WAKE_LOCK, keep the screen light and keyboard backlight always on. PARTIAL_WAKE_LOCK prevents the device from entering into the suspend state regardless of the screen's on-off states, by pressing the power button. Other WakeLocks are valid when the screen state is on, and do not affect the entry into the suspend state. Thus, if the user turns off the screen by pressing the power button, the device can enter the sleep state.

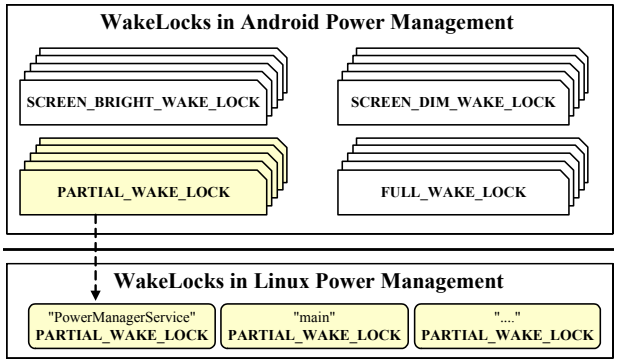


Figure 1. Management of WakeLock

Android power management is done via the *PowerManagerService* of the Android framework [5]. The service is responsible for all functionalities associated with the power-related features (i.e., WakeLock management, screen management, battery management, etc). Figure 1 illustrates the WakeLock mechanism related to Linux power management. Since multiple WakeLocks can be used from both the application and Android system, *PowerManagerService* tracks the WakeLocks separately for each case. If *PowerManagerService* receives a release request on the acquired WakeLock, the WakeLock tracking is terminated. When all the acquired WakeLocks are released, *PowerManagerService* enables Linux power management to enter the suspend state by switching the device from the awake to sleep state.

For PARITAL_WAKE_LOCK used by the application, the *PowerManagerService* delivers only one PARTIAL_WAKE_LOCK, called "PowerManagerService," to Linux power management regardless of the number of acquisition requests. *PowerManagerService* also passes the PARITAL_WAKE_LOCK, which occurs in the Android system, to Linux power management with a unique lock name. At this point, PARTIAL_WAKE_LOCK is delivered by using the /sys file system (*Sysfs*) between the *PowerManagerService* in the user-level and Linux power management in the kernel-level. Therefore, Linux power management continues tracking the PARTIAL_WAKE_LOCK from both the application and Android system according to a unique lock name and prevents the device from entering into the suspend state while the acquired WakeLock from the user level is maintained.

PowerManagerService does not deliver other WakeLocks except for PARTIAL_WAKE_LOCK to Linux power management since the goal of WakeLocks is to control the screen light and keyboard backlight. The WakeLocks perform these functions through the Android framework. The CPU is always turned on when the WakeLocks are acquired since Linux power management acquires PARTIAL_WAKE_LOCK, called "main," for ensuring interaction between user and the device while the screen is on. Therefore,

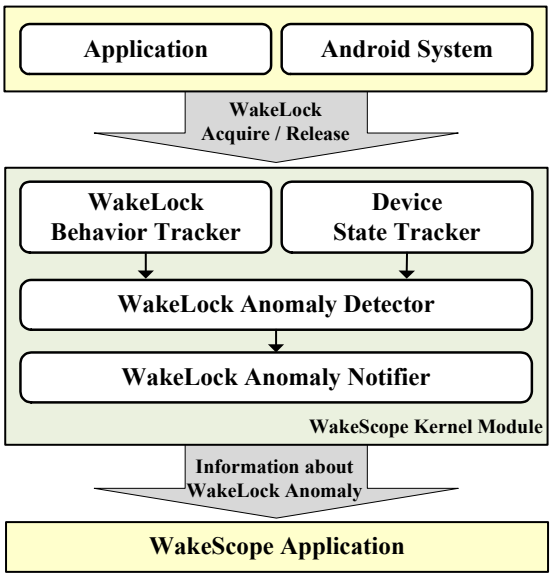
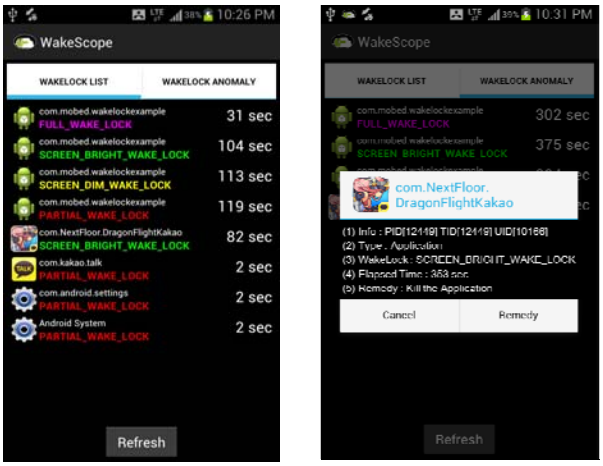


Figure 2. System Architecture of WakeScope



(a) WakeScope Application User Interface

(b) Dialog about WakeLock Anomaly

Figure 3. WakeScope Application

the WakeLocks are nullified when the users turn off the screen through pressing the power button. At this point, Linux power management releases the "main" PARTIAL_WAKE_LOCK and puts the device into the suspend state.

3. SYSTEM OVERVIEW

3.1 WakeScope Architecture

WakeScope accurately tracks the request and release of WakeLock usage both in the application and Android system, and detects the WakeLock anomaly, if it exists, in runtime. WakeScope also notifies a user of the anomaly, and provides an interface to handle the problem. Figure 2 shows the overall architecture of WakeScope. The scheme detects the misused case of WakeLock within the application and Android system at runtime. WakeScope is implemented as kernel modules which include WakeLock Behavior Tracker, Device State Tracker, WakeLock Anomaly Detector, and WakeLock Anomaly Notifier. The WakeScope Application is also provided for interaction with the user.

WakeLock Behavior Tracker traces the acquisition and release of WakeLock in runtime for both the application and Android system. The module tracks the WakeLock at the process-level where the process is classified into either application or Android system based on its UID (user id). Note that the current Android framework uses UIDs of around 1,000 for the Android system, and UIDs over 10,000 are used by the applications.

Device State Tracker traces the state information of the device that is used by the WakeLock Anomaly Detector. The tracked state information consists of screen on/off state, screen lock/unlock state, screen light off time and keyboard backlight off time. When the screen state is changed, the screen-related information is collected by monitoring the information about the screen state that is delivered to the Android *Binder* [6, 7] using *Kprobes* [8], which is a mechanism to hook kernel routines dynamically. Both the screen light off time and the keyboard backlight off time are transmitted from the WakeScope Application to the Device State Tracker through the /proc file system (*Procfs*).

The **WakeLock Anomaly Detector** finds a WakeLock anomaly from the WakeLocks tracked by the WakeLock Behavior Tracker. The process takes two steps: suspicion and detection. First, all the processes acquiring the WakeLock are examined by the WakeLock Anomaly Test (WAT). WAT checks if a process is running a critical job that must be performed without entering into the sleep state of the device until the acquired lock is released. If the running process after acquiring the lock stops without releasing it, the lock is suspected to be WakeLock anomaly. Second, the suspected WakeLock is finally determined to be an anomaly, based on the state information of the device obtained from the Device State Tracker. If the WakeLock type is PARTIAL_WAKE_LOCK, the anomaly occurs when the device screen is turned off. Otherwise, the anomaly is determined based on both screen light off time and keyboard backlight off time.

WakeLock Anomaly Notifier makes a user aware of the WakeLock anomaly when it happens. The information notified to the user includes the WakeLock type and the source of the anomaly (i.e., PID, TID, UID, name of the host). The /proc file system (*Procfs*) is used for the notification.

The **WakeScope Application** interacts between the user and the kernel module. Figure 3 shows the WakeScope Application. The tool lists the WakeLocks currently acquired and the one that is determined to be anomaly (Figure 3(a)). The tool receives a list of the WakeLocks from the WakeLock Behavior Tracker and WakeLock Anomaly Detector, respectively. It then notifies the user of the WakeLocks that are determined to be an anomaly through Android *Notification* [9]. WakeScope provides the functionality to handle the WakeLock anomaly (Figure 3(b)). The tool provides separate methods of handling the anomaly, depending on if the source is the application or Android system.

3.2 Challenges

WakeScope is an attempt to detect and manage WakeLock anomalies that may happen in the application and Android system in runtime. The scheme has to satisfy two requirements to meet its

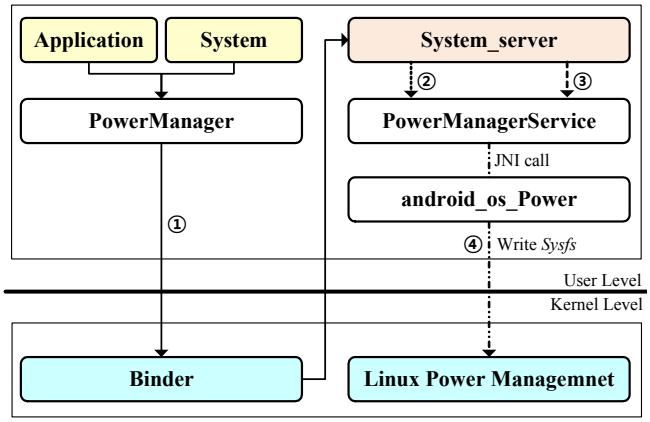


Figure 4. WakeLock Processing

goals. First, WakeScope should accurately track the WakeLock behavior from within the application and Android system. Inaccurate tracking of the WakeLock behavior can lead to incorrect decisions by the WakeLock anomaly. Second, the WakeLock anomaly should be accurately detected based on solid technical analysis.

4. WAKELOCK BEHAVIOR TRACKING

To track the WakeLock behavior in runtime, an efficient method should be devised. In Section 4.1, we analyze the WakeLock acquisition and release mechanism between Android power management and Linux power management. We then present the WakeLock behavior tracking process in Section 4.2.

4.1 WakeLock Processing

Figure 4 shows the overall process of WakeLock acquisition and release. The Android application and Android system work at the process-level in the Linux kernel. The Android system consists of the *system_server* process, which is responsible for the core operations of the Android framework (i.e., activity manager, power manager, location manager, etc) and other system processes. Applications run either in the foreground process in the form of an Android Activity or background process in the form of an Android Service. The Android power management service *PowerManagerService* works in the *system_server* and deals with requests for WakeLock acquisition and release from both the application and Android system. The requests are transferred to the *system_server* by using the Android *Binder RPC*, with which various types of messages are exchanged between processes, through the *PowerManager* proxy (Figure 4. 1). *system_server* delivers the request (Figure 4. 2) as well as its own request to *PowerManagerService* (Figure 4. 3). Depending on the requested WakeLock type, *PowerManagerService* uses a two-way process. In the case of PARTIAL_WAKE_LOCK, it calls on the *android_os_Power* module, which is responsible for communication with Linux power management through a JNI call, in order to deliver a request of the WakeLock. *android_os_Power* transfers the WakeLock from the *PowerManagerService* to Linux power management through the /sys file system (*Sysfs*) (Figure 4. 4). Other WakeLocks except for PARTIAL_WAKE_LOCK, control the device screen and keyboard through *PowerManagerService* depending on the WakeLock type.

4.2 WakeLock Behavior Detection

WakeScope detects the WakeLock behavior via monitoring kernel functions using *Kprobes*.

The requests for WakeLock acquisition and release from the application and Android system except for the *system_server*, are transferred to the *PowerManagerService* of the *system_server* through the Android *Binder RPC*. WakeScope monitors the *binder_transaction()* kernel function, which is the core of the Android's binder framework. WakeScope then analyzes the input parameter of the function related to the WakeLock behavior and detects the behavior if it exists.

The WakeLock usage originating from the *system_server* occurs inside the Android framework. The WakeLock behavior is not passed into the kernel, and thus it cannot be detected at the kernel level. As an alternative way to detect the WakeLock behavior, WakeScope uses a *Logcat* message [10] of Android. Since Android prints out the WakeLock behavior except for PARTIAL_WAKE_LOCK from the *system_server*, WakeScope monitors the *vfs_writev()* kernel function, which is used to print out the *Logcat* message by Android. WakeScope analyzes the message from the function related to the WakeLock behavior and detects the behavior if it exists. WakeScope detects the PARTIAL_WAKE_LOCK from the *system_server* by monitoring *wake_lock_store()* and *wake_lock_unstore()* kernel functions. These are the handler functions of the /sys file system (*Sysfs*), which are used to transfer the WakeLock from the Android platform to the Linux power management (Section 4.1).

The detected WakeLock is tracked at the process-level. The process using the WakeLock is decided by *Current* macro, which represents the currently scheduled process in Linux, when the WakeLock usage occurs from each kernel function. The process is classified into application and Android system based on its UID.

Android power management invalidates a request for the acquisition of the WakeLock from both the application and Android system if the request is duplicated. Also, Linux power management invalidates the PARTIAL_WAKE_LOCK request delivered from Android power management if the WakeLock is duplicated based on its lock name. Therefore, WakeScope invalidates duplicated requests for the acquisition of WakeLock during the process of WakeLock behavior detection.

5. WAKELOCK ANOMALY DETECTION

We now explain the method of WakeLock anomaly detection. The detection process is performed in two steps. When the state of the application and Android system, which have acquired the WakeLock, is changed from running state to stop state, WakeScope examines if WakeLock has been released. If the application and Android system stopped without releasing the WakeLock, a WakeLock anomaly would be suspected. Second, if the suspected anomaly is satisfied by certain conditions, the WakeLock is then determined to be an anomaly. In the following, we give full details of the WakeLock Anomaly Test (WAT).

5.1 WakeLock Anomaly Test

WAT decides the suspected WakeLock is an anomaly when the application or Android system that has acquired the WakeLock is stopped from completing its job, which must be performed without entering into the sleep state. Figure 5 illustrates the WAT process. WAT uses the current state of the Linux process and

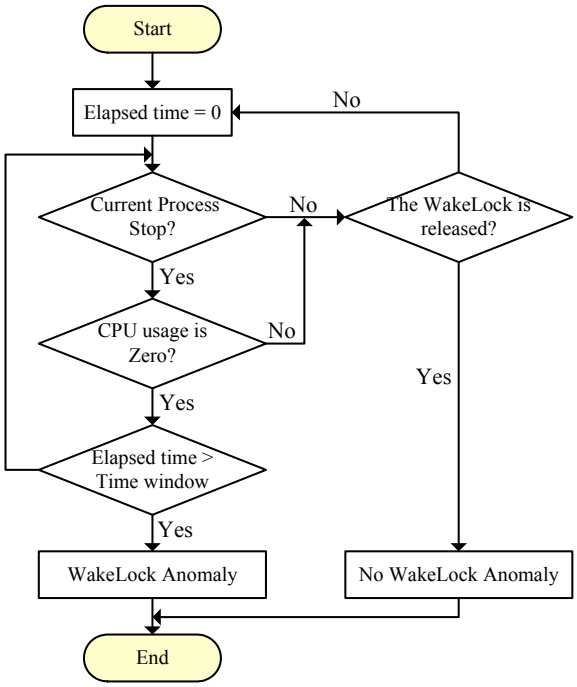


Figure 5. Process of WakeLock Anomaly Test

CPU usage to confirm the running and stop state of the application and Android system.

The application and Android system work at the process-level in the Linux kernel. We analyzed how the CPU usage and the state of the process changes depending on the behavior of the application and Android system. We observed three types of behavior. The first type is that the current state of the process is TASK_RUNNING and the process uses CPU continuously when the application and Android system perform a job according to a certain event, such as user interaction. At this point, CPU usage of the process is confirmed by the consumed CPU tick of the process via its *utime* and *stime*. Second, the current state of the process is TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE and the process uses CPU intermittently. In this case, the application and Android system are stopped until a certain event, such as user interaction or completion of the I/O operation, has occurred. Thus, the process uses CPU intermittently to wait for the event. Finally, the current state of the process is TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE and the process does not use CPU when the processes are terminated or switched into the background by the user. When the application and Android system are terminated or switched into the background, Android does not kill them; instead Android just saves their states and puts them into inactive status so that they can be reused by the user. Thus, the current state of the process is maintained with the above state and the process does not use CPU until they are reused. If they are killed by Android due to a lack of memory space, the current state of the process is changed to EXIT_DEAD.

Based on the above types, WAT checks the running and stop state of the application and Android system. The running state is determined based on the first and second types, whereas the stop state is determined based on the third type.

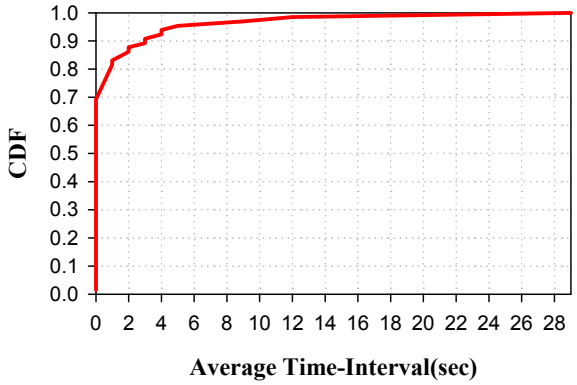


Figure 6. Cumulative Average Time-Interval distribution

In the case of the second type, WAT may misjudge that the current state is the stop state since the CPU usage occurs intermittently. Thus, WAT should wait for a certain amount of time until the CPU usage occurs. To check the intermittent occurrence of CPU usage, WakeScope assigns a time window to each process and WAT waits for this. In order to determine the window size, we analyzed the average time-interval for 90 applications and 10 Android systems that use a WakeLock. For the process that uses the CPU intermittently after acquiring the WakeLock, the average time-interval is the idle time between two consecutive CPU usages. As illustrated in Figure 6, 70% of them always use CPU until releasing the WakeLock and the rest take an average of 5 seconds or less. In the case of applications using GPS, we observed that it takes a maximum 29 seconds to receive location data from a GPS. Based on the result, WakeScope sets 60 seconds as sufficient for the default time window to confirm the intermittent occurrence of CPU usage by the application and Android system.

The applications and Android systems have different average time-intervals according to Figure 6. WakeScope has to set different time windows in every application and Android system. For this, WakeScope maintains a history of information, which is the maximum previous time-interval of each process and reduces the time window slowly until the previous value is approached. Algorithm 1 presents the pseudo-code for this scheme, which enables an adaptive time window depending on each application and Android system.

Likewise, if WAT discerns the suspected WakeLock is an anomaly, WakeScope determines if it is an anomaly based on certain conditions stated in Section 5.2.

5.2 Decision of WakeLock Anomaly

WakeScope has a different decision point to detect the WakeLock anomaly depending on the WakeLock type. PARTIAL_WAKE_LOCK prevents the device from entering into the suspend state. Linux power management also acquires PARTIAL_WAKE_LOCK, which has a lock name called “main,” for ensuring the interaction between user and the device while the device screen is kept on. Since a PARTIAL_WAKE_LOCK that is suspected as an anomaly can prevent the device from entering into the suspend state when the screen is turned off, WakeScope finally identifies it as a WakeLock anomaly. Other WakeLocks related to screen light and keyboard backlight are valid when the screen state is on and does not affect the entry into the suspend

Algorithm 1 Resize the time window

Input : Maximum average current time-interval T_{cur_max}, maximum average previous time-interval T_{prev_max}

Output : Resized time window T_{time_window}

```
if first time then
    T_prev_max ← T_cur_max
    return T_time_window
else
    if T_cur_max > T_prev_max then
        T_prev_max ← T_cur_max
        T_time_window ← T_time_window + 1 sec
        if T_time_window > T_default_time then
            T_time_window ← T_default_time
        end if
    else
        T_time_window ← T_time_window − 1 sec
        if T_time_window < T_prev_max then
            T_time_window ← T_prev_max
        end if
    end if
end if
return T_time_window
```

state of the system. Therefore, if the WakeLocks are suspected to be anomalies, WakeScope decides it when the amount of time that the screen light and keyboard backlight are on, according to the WakeLocks, is more than the screen light off time and the keyboard backlight off time of the device.

6. HANDLING OF WAKELOCK ANOMALY

When the WakeLock is detected as an anomaly, the user is notified and WakeScope handles it differently depending on if the source is the application or Android system. In the following, we present a method to handle the WakeLock, and then discuss an efficient method to deal with it.

For a WakeLock anomaly that is caused by an application, WakeScope simply kills the application. The Android framework then gets back all resources, such as memory, used by the application when the application is killed by the user or Android Low Memory Killer [11]. Since the WakeLock used by the application is included in the resource, if the application using WakeLock is killed, the Android framework releases all of the WakeLock acquired by the application. For the application case, the WakeLock anomaly occurs when the running application is stopped. In other words, the application that has generated the WakeLock anomaly is not currently used by the user. Therefore, it is reasonable to handle the WakeLock anomaly by killing the application. Meanwhile, the user may not want to kill the application. Thus, WakeScope interacts with the user concerning the WakeLock anomaly through dialog, as shown in Figure 3 (b). For the Android system case, WakeScope suggests a device reboot since the device would not work properly if the Android system is killed.

The best way of handling the WakeLock anomaly is to release the WakeLock without killing the application or rebooting the system. To achieve this, we may use the *goToSleep()* API [12] of the Android framework. This function forces the smartphone to go into the sleep state. At this point, all the acquired WakeLocks in the Android platform are released. Since WakeScope tracks all the WakeLock behavior in runtime, the WakeLock detected as an anomaly can only be released if the *goToSleep()* API is used when there are no WakeLocks acquired except for the WakeLock identified as an anomaly.

An inappropriate use of the *goToSleep()* API by an application developer can, however, cause malfunction of the device since the device is forcefully switched to the sleep state by the API. Thus, Android does not allow the *goToSleep()* API to be used by application developer in order to prevent such a situation. In summary, if *goToSleep()* API is allowed to be used, a more efficient method of dealing with WakeLock anomaly can be possible.

Table 2. Operation sequence of the test application

Name	Time (sec)	Acquire	Release	WakeLock Type	App Type
Case1	5	Thread A	Thread A	PARTIAL	Foreground
Case2	10	Thread A	Thread B	SCREEN_DIM	Foreground
Case3	15	Service	Service	SCREEN_BRIGHT	Background
Case4	20	Activity	Service	FULL	Foreground Background

7. EVALUATION

We evaluated WakeScope, focusing on its accuracy, overhead, and effectiveness. All evaluations were conducted on a Samsung Galaxy S3 [13] running Android 4.0.4 ICS.

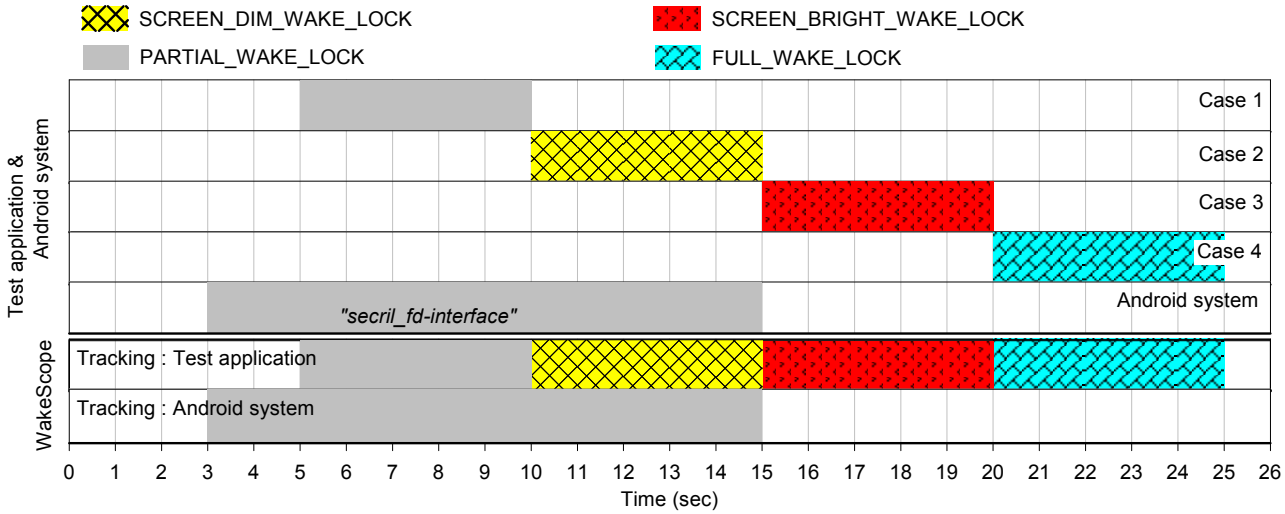
7.1 Accuracy

We evaluated the accuracy of WakeScope in two aspects: WakeLock behavior tracking and WakeLock anomaly detection.

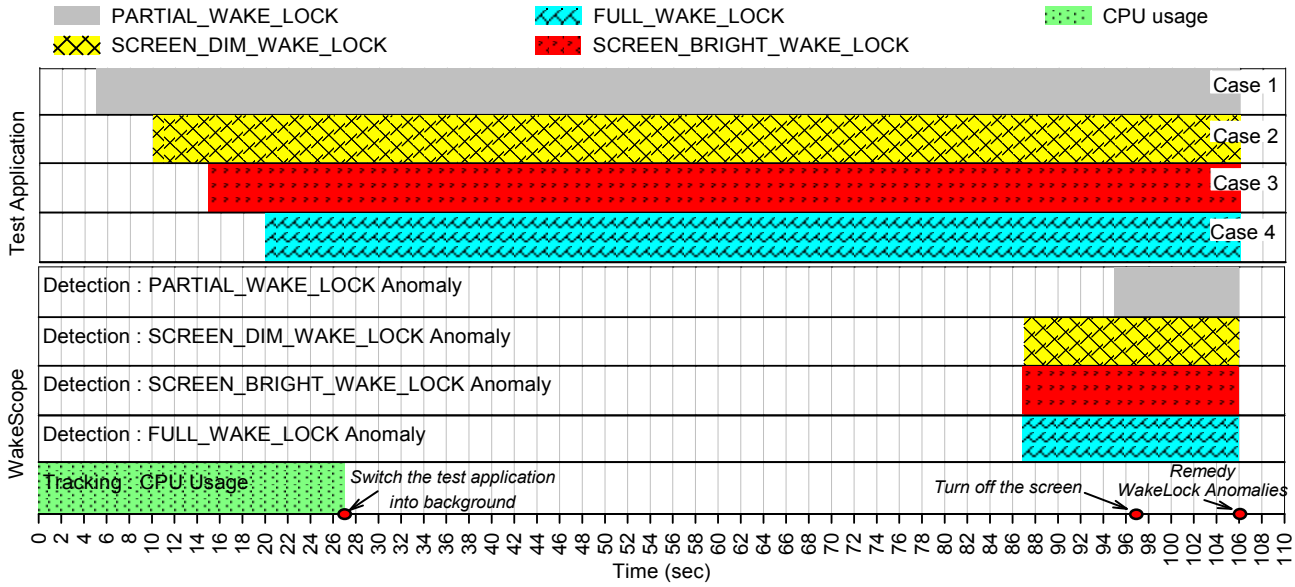
To evaluate the accuracy of WakeLock behavior tracking, we implemented a test application to emulate WakeLock behavior according to predefined scenarios. All possible scenarios of WakeLock acquisition and release are considered for the test application. Table 2[1] describes the test scenarios. Both Case 1 and Case 2 can occur by threads in the application and Android system. Case 3 represents the WakeLock behavior that arises with the Android Service and Android system working in the background. We designed Case 4 to allow the WakeLock behavior to occur in the foreground application and background application.

Figure 7(a) shows the results of WakeLock behavior tracking. For every case, the test application repeated the acquisition and release of WakeLock every 5 seconds. As shown in the figure, WakeScope accurately tracked the WakeLock behavior. During the test, we also observed that PARTIAL_WAKE_LOCK, which has a lock name called "secril_fd-interface," occurs in the

[1] Thread A and B in Table 2 mean different threads working in the test application.



(a) Accuracy of WakeLock behavior tracking



(b) Accuracy of WakeLock anomaly detection

Figure 7. Accuracy of WakeScope

Android system. This WakeLock is used to perform functionality related to the radio interface. We checked the "*/sys/power/wake_lock*" to confirm that WakeScope correctly tracked the WakeLock.

Meanwhile, to evaluate the accuracy of WakeLock anomaly detection, we modified the application to induce the anomaly situation by not releasing the WakeLock after acquiring it. We configured the screen off time and the keyboard backlight off time to 15 seconds and, 2 seconds, respectively. We also switched the test application to move from foreground to background when the application has completed the cases.

Figure 7 (b) shows the results of WakeLock anomaly detection. No CPU usage was detected by the application when switched into the background application at 27 seconds after completing all the tests. At this point, WAT detected the anomaly for the acquired WakeLocks. Since WakeScope set 60 seconds for the time window for checking the intermittent CPU usage, the anomaly can occur after 60 seconds, if one exists. The WakeLocks that occurred with Case 2, 3, and 4 were detected as an anomaly at 87 seconds since the elapsed time of WakeLocks by WAT was more than the screen light off time and the keyboard backlight off time. We turned off the device screen at 97 seconds. At this point, the PARTIAL_WAKE_LOCK anomaly, which occurred with Case 1, was detected at around 97 seconds. Finally, the WakeScope application was used to handle the anomaly at 106 seconds, then we observed that the anomalies had disappeared.

In summary, WakeScope is shown to track the WakeLock behavior and detect the anomaly accurately.

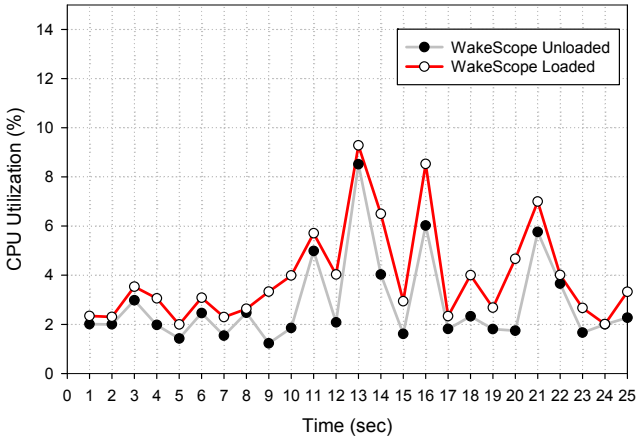


Figure 8. Overhead of WakeScope

7.2 Overhead

We analyzed the overhead of WakeScope. The overhead is mostly dependent upon CPU usage, as WakeScope primarily consumes the CPU resources.

To estimate the overhead of WakeScope, we compared the difference in CPU utilization while running the test application of Section 7.1 with and without WakeScope. We used the same test cases in Table 2. Figure 8 shows the CPU utilizations during the experiment. Overall, WakeScope consumed 1.2% more CPU resources, meaning WakeScope does not practically require additional CPU utilization to run all the scenarios in our experiment. We claim that WakeScope would exhibit small overhead, as it is not activated at all when no WakeLock behavior is used in the application or Android system.

7.3 Real Application Test

To evaluate the effectiveness of WakeScope, we observed the WakeLock anomaly with real Android applications. The anomaly occurs if an acquired WakeLock is not released when the application is stopped or terminated. We searched the anomaly through two user actions which are commonly conducted while using the device. The two actions are as follows:

The first action is to press the HOME key of the device while using the application. This action switches the application from foreground to background. That is, the application is stopped by this action; thus, the application should release the WakeLock until the application is reused.

The second action is to terminate the application which has been running in the background (i.e., the music player in this case). This action could be done through "Force Stop" in the Android setting. Since the application is forcefully terminated, the application should release the WakeLock.

Table 3 shows that WakeScope has indeed found three real world-applications that generate the WakeLock anomaly via the above two actions. **MyTracks** [14] is an application that records a moving path of the user in real time. The application acquires PARTIAL_WAKE_LOCK to record the user's moving path in the background without suspending until the recording is stopped. After starting the application, we applied the second action, and noticed that WakeScope detected the anomaly. We believe that

Table 3. WakeLock Anomaly and Overuse case

Application	Category	Downloads	Case
MyTracks	Life Style	1,000 – 5,000	Anomaly
DragonFlight	Game	10,000,000 – 50,000,000	Anomaly
Shooting Heros	Game	100,000 – 500,000	Anomaly
Littlecan	Game	1,000,000 – 5,000,000	Overuse
Ovoto	Social Contents	1,000 – 5,000	Overuse

the application developer did not release the acquired WakeLock when the application was terminated. Perhaps, the application developer should have released the acquired WakeLock in *onDestroy()* function [15], which is invoked when the application is terminated, according to the Android application's life-cycle. **DragonFlight** [16] and **Shooting Heros** [17] are two popular Android games. Once started, the applications acquire SCREEN_BRIGHT_WAKE_LOCK and FULL_WAKE_LOCK to turn on the screen and keyboard backlight, respectively, until the applications are stopped. After starting the applications, we applied the first action to the applications and noticed that, WakeScope reported the anomalies. Consequently, we observed that both screen and keyboard backlight were not turned off until the applications were correctly terminated or killed by Android. Since the energy consumption of the screen is the biggest part of total energy consumption [18], the anomalies are a very critical consideration form an energy perspective. As for the cause of the WakeLock anomaly, it was suspected that the application developer did not release the acquired WakeLock when the applications were stopped. The application developer should have released the acquired WakeLock in *onStop()* function [15], which is invoked when the application is stopped according to the Android application's life-cycle.

Meanwhile, WakeScope found two cases of WakeLock overuse, as Table 3 shows, where lock is unnecessarily used. **Littlecan** [19] is a popular mobile game running on the Android platform. Once started, the application acquires SCREEN_BRIGHT_WAKE_LOCK to ensure that the screen is always turned on. The *mediaserver* of the Android system acquires PARTIAL_WAKE_LOCK, which has a lock name called "audioOutLock" to perform audio-related operations for the application. At this point, we applied the first action, and observed that the acquired WakeLock was released. However, the acquired WakeLock by the *mediaserver* was not released. The reason is that the *mediaserver* was running without terminating the acquired WakeLock since the application did not stop the functionality when stopped. That is, the *mediaserver* was working meaninglessly with the WakeLock. **Ovoto** [20] is an SNS application that acquires PARTIAL_WAKE_LOCK. We observed that the application did not release the WakeLock when the application was terminated. This is because the application was not terminated successfully. Thus, the application was remaining in memory with the WakeLock and consuming the

CPU meaninglessly. In this case, energy would have been wasted due to the WakeLock overuse and meaningless CPU consumption.

8. RELATED WORK

Various attempts have been made to detect the WakeLock mishandling to prevent energy waste. Pathak et al. [3] defined the problem of WakeLock mishandling by an application developer as no-sleep bug. The WakeLock misuse was detected based on the path analysis of the no-sleep code and the cause of the problem was analyzed accordingly. Vekris et al. [4] defined a set of energy polices based on the Android life-cycle to detect the WakeLock mishandling. This work detects the WakeLock mishandling through static analysis with which the problem is analyzed at the source level at compile time. The work solves the problem from the developer view point. Our work shares the same goal, but we have a different approach. We detect and handle the problem in runtime, providing a practical solution for the user to prevent the energy waste caused by the WakeLock misuse.

There are many studies regarding the inefficiency of smartphone energy consumption. Pathak et al. [21] defined the type of energy bug that can be generated in the device, and analyzed their characteristics. Liu et al. [22] analyzed the energy inefficiency caused by the developer that uses sensors (i.e., GPS). Kim et al. [23] detected energy-greedy malwares in the device using the power signature of the device. eDoctor [24] studied the cause of abnormal battery drain in the device. Zhang et al. [25] compared and analyzed the cause of energy bugs in various smartphone platforms. Carat [26] analyzed data from a large number of users, detected energy anomalies, and suggested management solutions to users. Similar to prior work, our work focuses on the WakeLock mishandling problem, which may lead to energy inefficiency problems.

9. CONCLUSION

Careless use of the WakeLock mechanism could cause energy waste in Android-based smartphones since the device cannot enter into the sleep state. To prevent the problem, an effective scheme is needed to manage the misuse of WakeLock. In this paper, we presented a management scheme to handle WakeLock mishandling. Based on thorough analysis of the Android WakeLock mechanism, we successfully detected a misused case from both the application and Android system in runtime, and also provided a practical solution to handle the problem. According to our experiments, WakeScope guarantees the accurate detection of the WakeLock anomaly with low CPU overhead.

We believe that an efficient scheme to manage the WakeLock mishandling can be devised if our runtime solution fuses with compile time solutions suggested in prior work [3, 4]. Since WakeScope does not detect the problem at the source-level, our work cannot pinpoint the cause of the problem in detail. We expect that the misuse case of WakeLock can rightfully be managed by a hybrid system that analyzes the causes of the problem at compile time using prior work and also detects the problems with WakeScope in runtime.

AKNOWLEDGEMENTS

This work was supported by a grant from the National Research Foundation of Korea (NRF), funded by the Korean government, Ministry of Education, Science and Technology under Grant (No. 2013-027363).

10. REFERENCES

[1] "Android WakeLock mechanism" URL : http://developer.android.com/reference/android/os/PowerManager.WakeLock.html

[2] Android Types of WakeLock, http://developers.android.com/reference/android/os/PowerManager.html

[3] Pathak, A., Jindal, A., Hu, Y.C., P. Midkiff, S., 2012. What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps. In *Proceedings of 10th International Conference on Mobile Systems, Applications and Service* (Low Wood Bay, Lake District, UK, June 25-29, 2012). MobiSys'12. ACM, New York, NY, 267-280. DOI= http://dl.acm.org/citation.cfm?id=2307636.2307661

[4] Vekris, P., Jhala, R., Lerner S., Agarwal, Y., 2012. Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs. *In Proceedings of 2012 Workshop on Power-Aware Computing and Systems* (Hollywood, CA, October 7, 2012). HotPower'12, USENIX Association, Berkeley, CA.

[5] Android PowerManagerService, http://gitorious.org/android-eeepc/base/blobs/50108e9282031bbd3a22683061496285a806403e/services/java/com/android/server/PowerManagerService.java

[6] Android Binder, http://www.nds.rub.de/media/attachments/files/2011/10/main.pdf

[7] Openbinder, http://www.angryredplanet.com/~hackbod/openbinder/docs/html

[8] Kprobes, http://www.kernel.org/doc/Documentation/kprobes.txt

[9] Android Notification, http://developer.android.com/reference/android/app/Notification.html

[10] Android Logcat, http://developer.android.com/tools/help/logcat.html

[11] Android low memory killer, http://www.androidcentral.com/fine-tuning-minfree-settings-improving-androids-multi-tasking

[12] Android goToSleep(), http://developers.android.com/reference/android/os/PowerManager.html

[13] Samsung Galaxy S3 (SHV-e210S), http://www.samsung.com/sec/galaxys3feature/

[14] MyTracks, https://play.google.com/store/apps/details?id=jp.hiros.MyTracks&feature=search_result#?t=W251bGwsMSwxLDEsImpwLmhpcm9zLk15VHJhY2tzIl0

[15] Android activity life cycle, http://developer.android.com/training/basics/activity-lifecycle/starting.html

[16] DragonFlight, https://play.google.com/store/apps/details?id=com.NextFloor.DragonFlightKakao&feature=search_result#?t=W251bGws

MSwxLDEsImNvbS5OZXh0Rmxvb3IuRHJhZ29uRmxpZ2h0S2FrYW8iXQ

[17] Shooting Heroes, https://play.google.com/store/apps/details?id=com.xmongames.shootingheroes&feature=search_result#?t=W251bGwsMSwxLDEsImNvbS54bW9uZ2FtZXMuc2hvb3RpbmdoZXJvZXMiXQ

[18] Carroll, A., Heiser, G., 2010, An Analysis of Power Consumption in a Smartphone, *In Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (Boston, MA, June 23-25, 2010). USENIX ATC'10. USENIX Association, Berkeley, CA, 21-35

[19] Littlecan, https://play.google.com/store/apps/details?id=air.com.cjenm.littlecan&feature=search_result#?t=W251bGwsMSwxLDEsImFpci5jb20uY2plbm0ubGl0dGxlY2FuIl0

[20] Ovoto, https://play.google.com/store/apps/details?id=com.heystaks.ovoto&feature=search_result#?t=W251bGwsMSwxLDEsImNvbS5oZXlzdGFrcy5vdm90byJd

[21] Pathak, A., Hu, Y.C., Zhang, M., 2011. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. *In Proceedings of 10th ACM Workshop on Hot Topics in Networks* (Cambridge, MA, November 14-15, 2011). Hotnets'11, ACM, New York, NY. DOI= http://dl.acm.org/citation.cfm?id=2070562.2070567

[22] Liu, Y., Xu, C., Cheung, S., Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. *In Proceedings 2013 IEEE International Conference on Pervasive Computing and Communications* (San Diego, California, March 18-22, 2013). PerCom'13

[23] Kim, H., Smith, J., Shin, K., 2008. Detecting Energy-Greedy Anomalies and Mobile Malware Variants, *In Proceedings of 6th International conference on Mobile Systems, Applications and Service* (Brekenridge, Colorado, USA, June 17-20). MobiSys'08. ACM, New York, NY, 239-252. DOI= http://dl.acm.org/citation.cfm?id=1378627

[24] Ma, X., Huang ,P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L., Voelker, M., 2013, eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones, *In Proceedings of 10th USENIX Symposium Networked Systems Design and Implementation* (Lomard, IL, April 2-5, 2013). NSDI'13. USENIX Association, Berkeley, CA.

[25] Zhang, J., Musa, A., Le, W., 2013, A Comparison of Energy Bugs for Smartphone Platforms, *In Proceedings of 1st International Workshops on the Engineering of Mobile-Enabled Systems* (San Francisco, CA, USA, May 18-26, 2013). ICSE'13

[26] Oliner, A. J., Iyer, A., Lagerspetz, E., Tarkoma, S., Stoica, I. 2012, Collaborative Energy Debugging for Mobile Devices, *In Proceedings of 8th Workshop on Hot Topics in System Dependability* (Hollywood, CA, October 7, 2012). HotDep'12. USENIX Association, Berkeley, CA.