Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs

Panagiotis Vekris, Ranjit Jhala, Sorin Lerner and Yuvraj Agarwal
University of California, San Diego
{pvekris, jhala, lerner, yuvraj} @ cs.ucsd.edu

Abstract

The Android OS conserves battery life by aggressively turning off components, such as screen and GPS, while allowing application developers to explicitly prevent part of this behavior using the *WakeLock* API. Unfortunately, the inherent complexity of the Android programming model and developer errors often lead to improper use of WakeLocks that manifests as *no-sleep* bugs. To mitigate this problem, we have implemented a tool that verifies the absence of this kind of energy bugs w.r.t. a set of WakeLock specific policies using a precise, inter-procedural data flow analysis framework to enforce them. We run our analysis on 328 Android apps that utilize WakeLocks, verify 145 of them and shed light on the locking patterns employed and when these can be harmful. Further, we identify challenges that remain in order to make verification of Android apps even more precise.

1 Introduction

Smartphones have become pervasive over the last few years by incorporating features like multiple cores, digital cameras, GPS location tracking and large screens. To deal with resource management in the presence of these features, smartphone OSes are designed to aggressively put them to sleep as soon as they become "idle", while at the same time they allow developers to specify components they need to be kept awake. In particular, Android's WakeLock API enables developers to provide explicit resource management directives to the OS; acquiring a WakeLock object ensures that the device is on at the level specified at its creation [1].

WakeLocks provide a flexible and fine-grained resource management mechanism, but as they transfer the burden of their release to the developer, they have rendered apps prone to the so-called "no-sleep" bug, i.e. the situation where a resource (e.g. the screen, or the CPU) is kept awake indefinitely due to a misuse of the power control API [15, 17]. These bugs are particularly hard to detect since they do not lead to crashes or malfunctions, but rather reduce the device's battery life.

Pathak et al. [16] delineate the domain of no-sleep bugs in smartphone apps, characterize them completely and present the first compile-time technique for detecting them, based on the reaching definitions dataflow algorithm. Their tool reveals bugs in 44 out of the 86 tested apps, and provides a great first exploration of static analysis for no-sleep energy bugs.

In this paper, we further inspect this area by developing a new tool that verifies the absence of no-sleep bugs w.r.t. a set of policies. In doing so we make three contributions. First, we have studied the lifecycle of different kinds of components in Android applications, and use this information to precisely define a set of resource management policies specifying the correct usage of wake locks. Second, compared to prior work we apply more precise analysis techniques for handling asynchronous calls, which are ubiquitous in Android. Third, to evaluate our techniques we run our tool on 328 real world Android applications and verify that 145 of those are exempt from the specific kind of no-sleep bug (w.r.t. our policies). By analyzing warnings in the remaining applications we find energy bugs in several of them, and identify the important remaining challenges that must be addressed in order to precisely track energy usage in Android apps.

Related Work. Static analysis of Android applications is not new. Felt et al. [10] study Android apps to determine if developers follow least privilege with permission requests, and Chin et al. [8] provide a tool that detects vulnerabilities in the communication between apps. Security and privacy violations have also been studied [9, 11, 12]. Attempts to provide generic frameworks for app analysis include Ded [9], which is included in our tool's workflow, re-targets Dalvik bytecode to Java bytecode, and Dexpler [7], that converts Dalvik to Jimple, an intermediate representation used by Soot [5]. Eprof [17], a fine grained energy profiler for smartphone apps, revealed that

a considerable amount of energy is spent on third-party advertisement modules. Carat [14] detects and provides energy related recommendations, by collecting data from a community of devices. Pathak et al. [15, 16] investigate the pervasiveness of no-sleep bugs and present a tool that uses reaching definitions analysis to detect them.

2 Background

We start off with a brief description of Android's mechanism for power management and software components.

Power Management. To enable application-level power management, the OS allows applications to create, acquire and release `WakeLock` objects. A `WakeLock` is associated with a particular resource (CPU, screen, etc.), and when held prevents it from going into "sleep" mode. Thus, a developer can control the power state of the device by using the `PowerManager` class to create a lock on the relevant resource, acquire the lock while performing some critical task during which the device must stay on (e.g. while receiving updates from a remote server) and then release the lock after it is finished. However, *failure to release the locks in a timely fashion can lead to the device being kept on needlessly, thereby wasting power.* Android also has a timed acquire method, which allows the developer to specify a timeout after which the resource will be released automatically, but it is common for applications to eschew this mechanism in favor of manual management [1].

Application Components. An Android app is built from a set of *components*. These components are associated with a lifecycle whose stages correspond to the various parts of their functionality. The developer specifies the actions to be performed at each stage in the lifecycle by implementing a set of callback methods. There are four main kinds of components (outlined below), however, for the rest of this paper we are going to consider all application building blocks that execute in their own context, including Runnable objects, as components.

Activities provide the UI screen of an app. They form a stack, whose head is the currently *running* activity (in the foreground of the screen). An activity that is not in foreground, but still visible, is *paused*, and an activity in the background is *stopped*. Entry and exit from the running state is done by executing the `onResume` and `onPause` callbacks respectively, the paused state by executing `onStart` and `onStop`, and the entire activity's lifecycle is bounded by calls to `onCreate` and `onDestroy`. When a stopped activity is being restarted then the `onRestart` callback is called.

Services perform long-running operations without user interaction. A service can be *started* by calling `startService`, or *bound* upon by calling `bindService`, to offer a client-service interface to any interacting component. In either case, the service is set up during `onCreate`. `onStartCommand` is executed when a service is started by another component, and in the case of a bound service `onBind` and `onUnbind` will be called when the first and last client binds and disconnects from it, respectively. `IntentService` is a special type of service that handles asynchronous requests on demand, by spawning a separate worker thread and using it to carry out a task. Thus, at the end of `onStartCommand`, `onUnbind`, and `onHandleIntent` callbacks, the respective task should be completed and hence no wakelocks should be held.

BroadcastReceivers respond to system-wide broadcast announcements, from within the system (e.g. low battery), or from applications (e.g. to alert other apps that an event has occurred). A broadcast receiver begins and ends its work within the `onReceive` callback and hence, must not hold any wake locks at the end of this method.

ContentProviders provide the means to encapsulate and protect a structured set of data. Here, each exposed callback is typically a unit of work, and hence, all locks must be released by the end of the callback.

Intent-based Component Communication. Components can communicate via asynchronous messages called *Intents*, which offer a late run-time binding between components of the same or different apps. Intents can be *explicit*, i.e. targeted towards a component specified by its name, or *implicit*, i.e. not specifying a target, but an action to be performed. Precise accounting of inter-component communication is useful, as it is common for wake locks to be acquired within one component, and released within another component that is asynchronously invoked through an intent. The same holds regarding Runnable objects that can be explicitly spawned as threads or associated with a thread's *MessageQueue*.

3 Our Approach

To ensure that an app behaves well w.r.t. energy consumption we have developed a tool that statically verifies its compliance with a set of energy policies. In particular, our goal is to prove the absence of cases where there exists a path along which a wakelock is acquired but not released at the appropriate point in a component's lifecycle. To do this we first define a set of policies 3.1 and then we show how our analysis enforces them 3.2.

3.1 Policies

Energy States. No-sleep energy bugs arise when certain software components hold on to wake locks for an indefinite amount of time thereby forcing system resources to

[1] Uses of timed acquired wake locks are considered exempt from the no-sleep bug and are therefore not included in our analysis.

Component Type	Exit Point Callbacks
Activity	`onPause`
Service (Bound)	`onUnbind`
Service (Started)	`onStartCommand`
IntentService	`onHandleIntent`
BroadcastReceiver	`onReceive`
Runnable	`run`

TABLE 1: Main Locking Policies

remain awake and prevents them from descending to an idle state. Formally, we say a component is in a *high* (resp. *low*) energy state if it *is* (resp. *is not*) holding any wake locks at that point. Intuitively, our policies specify that at key *exit* points, where the component has finished doing work, the software component must be in a low energy state (i.e. must have *released* all wake locks.)

Exit Points. The behavior of an Android component is realized by a set of callbacks, which are invoked according to a lifecycle protocol that determines when the component is being setup, active, and shut down. Thus, to concretely specify the energy policy for a component, we need to map the callbacks to the phases in the lifecycle, in particular, to identify the callbacks *at the end of which* the component must be in a low energy state.

Component Policies. To identify the exit points, we partition components into categories, for each of which we have identified the callbacks whose exits must be in low-energy states. The categories and callbacks are summarized in Table 1. Note that components lacking a well-defined lifecycle, or not hard-coded in our analysis, are treated conservatively by requiring *all* of their callbacks to end in a low-energy state.

Asynchrony. Asynchronous transfer of control flow is ubiquitous in Android software. In particular, a common pattern (that was empirically established) is that of a component acquiring a wake lock on a resource and then asynchronously calling another component (e.g. through the *intent* mechanism), thereby implicitly delegating the responsibility of releasing this WakeLock to the latter. In these situations, the original component is in a high-energy state, but the program is energy-safe as long as the asynchronously invoked component releases the WakeLock by the time it reaches its exit point. However, note that if the triggered component does not operate on the WakeLock at all, then responsibility for releasing it remains with the original calling component.

3.2 Verification

We employ flow- and context- sensitive inter-procedural dataflow analysis [18] to verify that components adhere to their energy policies. Next, we describe how we track the energy state via dataflow facts, how we statically propagate the flow-facts via abstract transfer functions, and our inter-procedural framework for precisely analyzing synchronous and asynchronous method calls.

Dataflow Facts. Our analysis statically tracks the energy state at each program point via dataflow facts that represent the *set of* WakeLock instances that are held (i.e. acquired by the program) at that point. The program is in a low-energy state iff the lockset is empty.

Inter-procedural Control Flow Graphs (ICFG). We use a dataflow analysis to compute the lockset at each program point, and hence, identify the set of points where the program is in a high energy state. Our analysis represents the program with an inter-procedural CFG whose vertices are program points and edges are program instructions. Each method has a distinguished entry and exit vertex. Regular method calls are modeled by call- and return-edges between the call-site and the entry and exit points. As discussed earlier, we have found that verification requires precise modelling of the application lifecycle and analysis of asynchronous calls that delegate lock-release responsibilities. We carry out several auxiliary analyses to extract this information and reflect it in the ICFG.

Asynchronous Calls. Asynchronous calls are triggered by intents and thread spawning. To identify the *targets* of these calls – namely the component that will be executed or the target of a Runnable call or post – we use a standard intra-procedural *def-use* analysis, in which we track the definition sites of the parameters that are passed to the asynchronous call instruction to infer the target component. We only handle explicit intents in which a single target component is specified. Note that failure to resolve an asynchronous call, even at a high-energy state, leads to imprecision (not unsoundness) as in this case, the verifier will raise a warning if it was the (undetermined) target that released the WakeLock on the caller's behalf. Furthermore, the application can still be verified if the WakeLock is released later by the calling component itself.

Lifecycles. Recall that each component's callbacks are invoked according to a lifecycle protocol which specifies the order in which the callbacks may be invoked. The protocol is represented by a state machine whose vertices are the callbacks and edges denote the successor callback. To model the notion of a component's lifecycle, the *exit* ICFG node of a callback method is connected to *every entry* CFG node of each of its successor method in the component's lifecycle. These additions allow us to operate on a single CFG that encompasses the notion of an application's lifecycle and which also captures asynchronous inter-component communication.

Dataflow Analysis. After building the ICFG, we use the inter-procedural analysis [18] module of the WALA framework [6] to compute the lockset at each program point. We provide a *transfer function* to propagate flow facts across the edges of the ICFG: lock facts are *gen-*

Component	Callback	#Violating apps
Activity	`onPause`	100
	`onStop`	90
BcastRcvr	`onReceive`	53
Service	`onStart`	16
	`onUnbind`	28
	`onHandleIntent`	3
Runnable	`run`	30
AsyncTask	`onPostExecute`	5
other	*any*	43

TABLE 2: Violation report after examining 328 apps.

erated at `WakeLock.acquire` sites, they are propagated unchanged across normal edges, and *killed* at calls to `WakeLock.release`. Facts are propagated at return edges in the usual inter-procedural manner. At a *join* point, the set of locks held is the union of the locks at the predecessor points. At asynchronous calls and returns, we tag and untag the set of WakeLocks to track the set of components that have operated on each WakeLock, thereby allowing us to precisely identify which components were responsible for releasing the lock.

Policy Checking. Once we have computed the dataflow solution, we determine the category of the component by traversing the class hierarchy to find the appropriate component superclass. We then check whether the exit points specified by the corresponding policy are indeed at a low-energy state (as determined by the dataflow analysis). If so, the component is verified to be energy safe, and if not, we flag a warning.

4 Evaluation

To apply our policies on real-world apps, we downloaded 2,718 free apps from Android Market and freewarelovers.com. Of these 2,718 apps, we focused on the 740 (27.2%) which used the WakeLock API. We extracted the content of the .apk file of each of these apps and converted it to Java bytecode using tools such as ded [13], Soot [5] and dex2jar [2]. Unfortunately, due to limitations of these tools, we were only able to convert 328 of the 740 applications into Java bytecode (44.3%).

We ran our analysis on these 328 apps and verified that 145 (44.2%) comply with our policies. For the remaining 183 (55.8%), our analysis was not able to show that our policies hold. We manually inspected a set of 50 randomly selected apps (out of 183), and we were able to both find bugs (which we describe first), and also sources of imprecision in our analysis (which we cover second).

Common Bugs. Table 2 illustrates our analysis' results, based on the components and the callbacks where policy violations occur. The most common violation occurs in the implementation of activities using WakeLocks. Developers often fail to release a lock on a resource when they override the `onPause` or `onStop` callback, thus leading to the corresponding resource remaining held even after the user has navigated away from the activity screen (by hitting "Home" or by triggering a new activity). Several apps by Imoblife Inc. [3] featured this pattern, and so by installing these apps on a test Android device, we could verify using the Android debugging shell that a Partial WakeLock (that ensures that the CPU is running) was kept alive after powering up the application's main screen and then navigating to the home screen. In addition, a considerable number of apps were flagged as violating our policy regarding BroadcastReceivers. A BroadcastReceiver object is only valid while `onReceive` is executing and hence our tool flags situations where a WakeLock is held beyond the execution of this callback. Similar violations to policies pertaining to services' lifecycle were also triggered, as well as for other components that could not be determined during component resolution.

Asynchrony. To justify our decision to track asynchronous intent calls, we ran our tool with asynchronous call resolution disabled. This caused 5 of the applications that were earlier verified to be rejected. Note, this number would be higher if our results were not skewed by precision issues (mentioned later). NetCounter [4] (version 0.14.1) was one of these cases; the `onReceive` method of the `OnAlarmReceiver` BroadcastReceiver starts the `NetCounterService` after acquiring `mLockStatic` (a partial WakeLock). The target service releases the WakeLock as soon as its `onStart` method is called, and so this lock pattern is not considered to be harmful by our asynchronous-call-aware analysis. A different approach, on the other hand, that ignored the service call, flagged the application as harmful, as high energy state reached the end of the `onReceive` callback. We note that a conservative approach, like the one proposed by Pathak [16], that would account for all possible targets of the service call would not be able to verify this case, as we cannot expect all possible target-services to release the WakeLock in question as soon as they execute.

Causes of Imprecision. As with most verifiers, failure of our tool to verify that an app satisfies our policies, does not necessarily mean that the app is buggy. This is due to precision issues inherent to static analysis, leading to *false positives* (i.e. the app is correct, but our tool cannot verify it). The most important of these issues is the lack of full path sensitivity. For example, our analysis cannot handle cases where the acquire and release operations are guarded by the same condition, or when the lock is only acquired/released under some complex condition. Additional imprecision is introduced when wrapper methods, controlled by parameters, are used around acquire/release operations. The latter issue can be resolved by enhancing our tool with a constant propagation analysis. Further-

more, although our analysis has some path sensitivity on special conditionals, i.e. it propagates a "released" state at the false branch of an `WakeLock.isHeld()` or a "non-null WakeLock object" check, this path sensitivity can be fooled if the developer manually re-implements the functionality of `WakeLock.isHeld()` using other primitives.

Intent resolution also causes imprecision. Since the target component of an intent is an object, intent resolution reduces to static alias analysis, which is known to be a hard problem. We use standard alias analyses from WALA, which inevitably suffer from imprecision (about 60% success rate for intent resolution), causing intent calls to be ignored, which then leads to false warnings.

Soundness. A current limitation of our analysis arises from the treatment of exception edges. WALA's support for exception handling is rather conservative and imprecise: exception edges that could flow out of a series of instructions (e.g. field access) are not connected to the nearest relevant catch block but to the exit node of the CFG. Propagating state through these edges would be a huge source of imprecision for our analysis, as even if the developer did actually release a WakeLock in the `catch` or `finally` block of a `try` block, high energy state would circumvent it. Instead, we decided to kill all facts over exceptional edges, with the expectation that a more precise CFG inference algorithm with respect to exception handling would fix this limitation.

An important aspect to consider is the correctness of our policies. During their design we focused on the components that made most use of the WakeLocks, i.e. Activities, Services and BroadcastReceivers. Our policies are, therefore, more refined for these components and more coarse-grained for the rest. Further, regardless of the type of component they are related to, all our policies refer to the exit block of certain callback methods which leads to both precision and soundness issues. Precision is lost since we ignore any notion of lifecycle for our so called "unresolved" components and instead follow the more conservative approach of examining the exit state of every possible callback. Finally soundness is jeopardized, as our static technique does not track how long a callback takes to execute, and so by examining just the return point of a method, it overlooks the case of a method looping and not terminating (a situation akin to no-sleep dilation [16]).

5 Discussion

The main contribution of this paper is a tool that verifies that an Android application abides by a set of policies regarding its use of the WakeLock API. Running our analysis on a total of 328 Android applications revealed a considerable number of bugs (some of them verified manually) and enlightened us on the use of resource management principles in real-world applications.

Our long-term goal is to create a full-fledged analysis that verifies the absence of all no-sleep bugs. Some of the steps that remain towards this goal are to: (1) formally define a set of policies to fully express the conditions that need to be held to guarantee the absence of all kinds of no-sleep bugs, as described in [16] (e.g. including no-sleep dilation); (2) combine these policies in our existing analysis or extend the analysis if necessary; (3) address the precision and soundness issues outlined above.

Acknowledgements We thank John McCullough for his help in organizing our test database and Dimitar Bounov for his useful comments. We also thank our shepherd and our anonymous reviewers. This work was supported in part by NSF grants SHF-1018632, CCF-1029783.

References

[1] Android Developers. `http://developer.android.com/`.

[2] dex2jar. `http://code.google.com/p/dex2jar`.

[3] Imoblife Inc. `http://downloadandroid.info/`.

[4] NetCounter. `http://www.jaqpot.net/netcounter`.

[5] Soot. `http://www.sable.mcgill.ca/soot`.

[6] T.J. Watson Libraries for Analysis. `http://wala.sf.net`.

[7] BARTEL, A., KLEIN, J., LE TRAON, Y., AND MONPERRUS, M. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *SOAP* (2012).

[8] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in android. In *MobiSys* (2011).

[9] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *USENIX Security* (2011).

[10] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *CCS* (2011).

[11] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, vol. 7344 of *LNCS*. 2012.

[12] GILBERT, P., CHUN, B.-G., COX, L. P., AND JUNG, J. Vision: automated security validation of mobile apps at app markets. In *MCS* (2011).

[13] OCTEAU, D., ENCK, W., AND MCDANIEL, P. The Ded Decompiler. Tech. Rep. NAS-TR-0140-2010, Network and Security Research Center, Pennsylvania State University, Sept. 2010.

[14] OLINER, A. J., IYER, A. P., LAGERSPETZ, E., TARKOMA, S., AND STOICA, I. Carat: Collaborative energy debugging for mobile devices. In *HotDep* (2012).

[15] PATHAK, A., HU, Y. C., AND ZHANG, M. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In *HotNets* (2011).

[16] PATHAK, A., HU, Y. C., AND ZHANG, M. What is Keeping my Phone Awake? Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *MobiSys* (2012).

[17] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the Energy Spent Inside my App? Fine Grained Energy Accounting on Smartphones with Eprof. In *EuroSys* (2012).

[18] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *POPL* (1995).