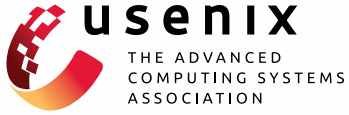


Selectively Taming Background Android Apps to Improve Battery Lifetime

Marcelo Martins, ***Brown University;*** **Justin Cappos,** ***New York University;***
Rodrigo Fonseca, ***Brown University***

https://www.usenix.org/conference/atc15/technical-session/presentation/martins

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

Selectively Taming Background Android Apps to Improve Battery Lifetime

Marcelo Martins
Brown University

Justin Cappos
New York University

Rodrigo Fonseca
Brown University

Abstract

Background activities on mobile devices can cause significant battery drain with little visibility or recourse to the user. They can range from useful but sometimes overly aggressive tasks, such as polling for messages or updates from sensors and online services, to outright bugs that cause resources to be held unnecessarily. In this paper we instrument the Android OS to characterize background activities that prevent the device from sleeping. We present Tamer, an OS mechanism that interposes on events and signals that cause task wakeups, and allows for their detailed monitoring, filtering, and rate-limiting. We demonstrate how Tamer can help reduce battery drain in scenarios involving popular Android apps with background tasks. We also show how Tamer can mitigate the effects of well-known energy bugs while maintaining most of the apps' functionality. Finally, we elaborate on how developers and users can devise their own application-control policies for Tamer to maximize battery lifetime.

1 Introduction

The accelerated growth of sensing, computational, storage, and communication capabilities of mobile devices has enabled a rich application environment that rivals the performance of desktop computers. Even so, battery technology has not followed the same advancement pace and there is little evidence that this situation will dramatically improve. As a result, battery lifetime has become a major usability concern, with users willing to enjoy the latest apps on their smartphones and tablets, but, at the same time, worrying that their battery will not last long enough.

Since the inception of mobile computing, both industry and academia have developed a slew of techniques to reduce power at the architecture [8, 24], OS [49, 43] and application levels [16, 21], and today's systems draw little power while idling. Due to its user-centric and interactive nature, the flow of a mobile application is driven by events such as user actions, sensor I/O, and message exchanges. Such event-driven paradigm lets the system idle until a new event arrives. Mobile OSes, such as Android, iOS, and Windows Phone, take advantage of such idling opportunities to engage in *opportunistic suspend*. Upon brief periods of idling, the handheld switches to the default suspend state. Hardware blocks, including the CPU, GPU, GPS, and network modem, shift to low-power mode and software state is kept in self-refreshing RAM. The same blocks return from suspension upon interrupts emitted by hardware or software indicating that they have pending requests. With the rise of multitasking and the multiplication of background services and complex mobile applications, we expect this amount of interrupts to increase, forcing the system to spend more time *active* to attend requests. Such active periods take a toll on battery lifetime. A recent study by Google clearly shows this impact: each second of active use of a typical phone reduces the standby time by two minutes [18].

This paper studies the problem of battery drain mostly due to app-originated background operations that wake up the mobile system. We present Tamer, an OS mechanism we built for Android that interposes on events and signals responsible for task wakeups – alarms, wakelocks, broadcast receivers, and service invocations. Like a number of profiling tools, Tamer allows us to characterize the background behavior of different apps installed on a device. In §2, using Tamer's instrumentation, we show how a set of installed applications can dramatically affect the battery lifetime of four different devices. Unlike existing profiling tools, however, Tamer can also selectively block or rate-limit the handling of such events and signals following flexible policies. In this way, Tamer can, for example, limit the frequency at which an application schedules alarms or receives notifications of specified events, providing fine-grained control over the energy usage of apps that may be useful, but are irresponsible or inefficient with respect to their background activities. In §5.2 we show via a few case studies how Tamer can reduce the energy consumed by energy bugs [35] in legitimate apps. We summarize our contributions as follows:

- We characterize how applications and core components of the Android OS use specific features to enable background computing, and how this computing significantly affects energy use. In special, we note that Google Mobile Services play a major role on battery drain while the device is dormant (§2).
- We introduce Tamer, an OS mechanism to control the frequency at which background tasks are handled, thereby limiting their impact on energy consumption (§4). Tamer leverages code-injection technology and is applicable to any Java-based Android application.
- We demonstrate how Tamer can successfully throttle the background behavior of popular applications, thus reducing their energy footprint (§5). We show

how different policies reduce power draw in exchange for little to no impact on functionality.

Despite being a powerful mechanism, Tamer is only a step towards effective control of background energy usage. In particular, there are still challenges in helping users define policies that are effective, yet not disruptive to the user experience. We discuss such challenges in §7.

2 Motivation

Smartphone and tablet users are used to being always connected, expecting immediate notifications of a new e-mail or application update. Other common background operations include polling navigational sensors for location clues and turning on the network radio for incoming messages. Especially in Android, where there is little restriction on what apps can do in the background and developer's discipline is the main factor preventing inefficient applications, apps can hog resources and waste energy.

Traditionally there has been little visibility, both to app developers and to users, on the contribution of individual apps to energy use, especially while in the background. It is even harder to know whether an app is actually running or idling. Recent monitoring and profiling tools have helped bridge this visibility gap [9, 39, 19, 36, 30, 33, 45], as one cannot optimize what cannot be measured.

Today's average handheld contains a large amount of third-party software. A 2013 survey shows a global average of 26 apps installed on a mobile device [20]. Even with the best currently available tools, the end user can do little to cope with inefficient apps. Most of the tools above target developers and provide little help for the user. Even the friendlier ones, such as eStar [30] and Carat [33], when highlighting energy-inefficient programs, can only offer to kill or uninstall the culprit app, perhaps suggesting replacements. Unfortunately, this is too coarse-grained a solution and some apps with irreplaceable functionality become an inconvenience one has to live with.

Tamer offers the possibility of much finer-grained control once an energy hog or bug is found. It provides information on which tasks are expending the most energy and can rate-limit their execution. Tamer detects most causes of device wakeup that are visible at the framework level of Android, and can filter their continuation in real time.

To demonstrate the significant difference that a set of running tasks can make in a device's battery life, we measured the battery drop of four Android devices (two smartphones and two tablets, cf. Table 1) running two different application sets, *while idling and with the screen off*. Conservatively, we consider three scenarios: the first testing environment (Pure AOSP) consists of a stripped version of the Android Open Source Project (AOSP) OS containing a minimum number of services and apps; the second one adds Google Mobile Services (GMS) on top of Pure AOSP. GMS consists of proprietary applications and services developed by Google, such as Calendar, Google+ (social media), Google Now (personal assistant), Hangouts (instant messaging), Maps, Photos, Play Service (integrating API), Play Store, and Search. Due to their popularity and added value, GMS apps are included in most Android devices sold today. For the third scenario, which we only ran on the Galaxy Nexus phone, we also installed the ten most popular free apps of Google's Play Store as of January 2015[1]. We based all environments on the KitKat (4.4) version of Android. For the experiments, we left each device unattended running with its configuration at default settings. Other relevant settings include connection to a WiFi access point, enabled location-reporting, and background network synchronization. We expect most of the battery drainage to stem from static-voltage leakage and eventual background processing.

Figure 1 shows the time taken by each environment-device combination to deplete the battery. For all devices, Pure AOSP took the longest to completely drain the battery. In the case of tablets, this difference spanned dozens of hours. To investigate why this happened, we instrumented the Android software stack to timestamp the occurrence of background events. Additionally, we connected one of our devices (Galaxy Nexus) to a Monsoon power monitor [31] and collected power traces from the battery. Finally, we aligned and synchronized both the event and power timelines to understand their correlation. Figure 2 depicts a six-minute slice of this combination. We observe that GMS triggers more events in the background and that they are correlated with the surge of power peaks. We used this tracing knowledge to build a mechanism that counters the energy effect due to excessive wakeups. Because this mechanism relies on OS internals, we first need to understand how an Android app functions while in the background.

3 Background

This section provides a concise description of Android's power-management system followed by an overview of the components constituting a mobile app and how applications behave while running in the background. Finally, we highlight the influence of background execution on battery drain using four types of events: wakelocks, services, broadcast receivers, and alarms. §4 describes Tamer, our control system that adjusts the frequency at which such events occur.

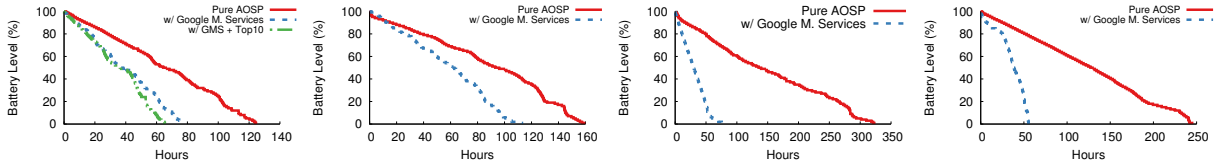
3.1 Mobile Power Management

Android employs an aggressive form of power management to extend battery life. By default, the entire system suspends itself, sometimes even when there are processes running. Opportunistic suspend is effective in preventing programs from keeping the system awake and quickly

[1] Crossy Road, Candy Crush Soda Saga, Pandora Radio, Trivia Crack, Snapchat, Facebook Messenger, Facebook, 360 Security Antivirus, Instagram and Super-Bright LED Flashlight.

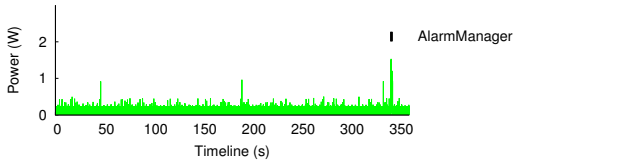
Device Name	Device Type	Processor	Features
Google Galaxy Nexus	Smartphone	Dual-core 1.2GHz ARM Cortex-A9	WiFi, GPS+A-GPS, 3G
Samsung Galaxy S4	Smartphone	Quad-core 1.9GHz Qualcomm Krait 300	WiFi, GPS+A-GPS, 3G/LTE
Amazon Kindle Fire 2	Tablet	Dual-core 1.2GHz ARM Cortex-A9	WiFi
ASUS MeMO Pad 7 (ME176C)	Tablet	Quad-core 1.83GHz Intel Atom Z3560	WiFi, GPS+A-GPS

Table 1: List of devices used for battery-drop monitoring.

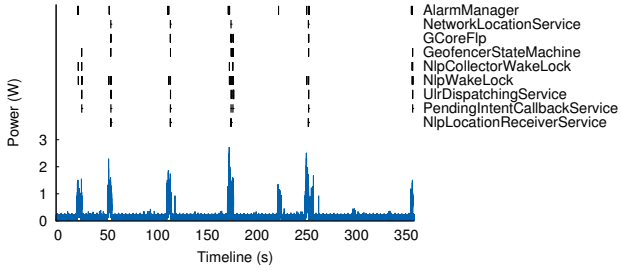


(a) Galaxy Nexus (b) Galaxy S4 (c) Fire 2 (d) MeMO Pad 7

Figure 1: Battery drop of four Android devices idling with the screen off. With Google Mobile Services installed, battery life decreased to 29.5% (Fire 2) and 77.5% (MeMO Pad 7) of its initial decay (without GMS).



(a) Pure AOSP



(b) Pure AOSP + GMS

Figure 2: Six minutes sampled from measurements on the Galaxy Nexus for two scenarios. For each graph, the top stack shows different event occurrences over time. The bottom curve depicts the corresponding system's power draw during the same period.

draining the battery. To curb system suspension, Android uses `Wakelocks` to keep the system awake. Wakelocks are reference-counted objects, similar to concurrency locks, that can be acquired/released by kernel and privileged userspace code. A wakelock acquire expresses a process's need for the system to remain awake until run completion. A wakelock acquire either holds a resource awake until a release call occurs or sets up a timer to relinquish the lock at a later time.

Kernel drivers use wakelocks to block the suspension of different system components (e.g., CPU, network, screen), whereas the Android application framework leverages wakelocks for different levels of suspension, represented by groups of components (e.g., keep the network radio awake vs. keep the radio, screen, and CPU awake). As an example of suspend-blocking by the OS, Android automatically acquires a wakelock as soon as it is notified of an input event and only releases the wakelock once some application handles the event or there is a timeout. Application developers can also directly instantiate and manipulate wakelocks using the Wakelock API. A proper e-book reader app must acquire a wakelock to keep the screen awake so that the user can read her favorite novel without interruption. Wakelocks play an important role in guaranteeing proper background task execution in face of default suspension, as we will see next.

3.2 Android Applications: Dealing with Lifecycle Changes

Barring a few interface-less system processes, an Android application consists of a set of `Activities` that places the UI widgets on the screen. An application starts with a single thread of execution attached to the *foreground* UI, which is mostly responsible for dispatching interface events. To avoid app unresponsiveness and user frustration, a wise programmer would move other computations to concurrent worker threads while the UI responds to input events. Support for concurrency comes in the form of a number of standard Java primitives, such as `Threads` and `Futures`, as well as Android's own flavors: `AsyncTasks` and message `Handlers`. However, such primitives only work when the application is in the *active state*.

As the user navigates through, out of, and back to an application, its lifecycle transitions between different states according to activity visibility. An application is active if one of its activities receives user focus in the foreground. If the user switches to another app or decides to turn off the screen, the application is paused and moved to the background. Because mobile apps are multitasked, developers must have a way to run code even when their app is not occupying the screen.

3.2.1 Dispatching Background Tasks

The small screen size of a smartphone or tablet prevents multiple applications from running simultaneously. To conserve energy, apps are frozen and stop working once sent to the background, either due to the opening of another application or a screen timeout. Context switching opens room for opportunistic suspend – by making apps invisible, the Android OS frees its own set of wakelocks, opening space for hardware throttling.

Android offers application developers a small and well-defined interface for background-task offloading that takes care of scheduling latent tasks [2]. This interface comprises a handful of components including services, broadcast receivers, and alarms. The internal implementation of such components also leverages wakelocks to keep the device awake while executing tasks.

`Services` are application components that run asynchronously on background threads or processes and do not directly interact with the user. Instead, `Activities` dispatch services to perform long-duration operations or to access resources on behalf of users, such as downloading remote files or synchronizing data with a cloud-based storage. An advantage of running services separately is that their running persists even after closing the owner's interface. Apps and widgets rely on services being operational without need of manual restart.

`BroadcastReceiver` is a reactive mechanism that permits programs to asynchronously respond to specified events. An application registers a `BroadcastReceiver` along with an event-subscription list – the `IntentFilter` – that is used to determine if the application is eligible to respond to a given event. Events can be predefined by the system (e.g., "battery fully charged") or developer-defined (e.g., "backup finished"). Receiver threads remain dormant until a matched event arrives and respond by running a callback function. A file-hosting app could, for instance, register a receiver to display a notification box once it discovers that a scheduled data synchronization has finished.

Another common programming pattern is the ability to perform time-based operations outside the lifetime of an application. For instance, checking for incoming e-mails every so often is a recurrent user operation that could be automatized. The Android SDK offers developers the `AlarmManager` mechanism to fulfill the scheduling of periodic tasks at set points. At each alarm trigger, the system wakes up and executes the scheduled callback function, whose contents can take various forms: a UI update, a service call, an I/O operation, scheduling a new alarm, etc. Alarms are a good fit for opportunistic suspend: apps are only activated when there is work to do.

In summary, Android uses at least three types of asynchronous mechanisms to perform background tasking: services, broadcast receivers, and alarms. Aligned with wakelocks, we have a powerful collection of events that can keep the system awake. In the next section, we introduce Tamer, a system that acts on this small and well-defined interface to throttle the rate at which background events are handled in exchange for energy savings.

4 Tamer

4.1 Design

Having seen that background events can noteworthily affect the sleeping pattern of mobile devices, we consider the possibility of regulating their frequency to improve battery life. We introduce a configuration mechanism for declaring thresholds to the frequency of these events. We model this regulation process using three sequential steps: (1) observation; (2) comparison; and (3) action.

Event-frequency regulation works based on the specification of occurrence limits. We establish a policy mechanism that lets users define how often the running system should permit a given background event to proceed. A policy is a contract that declares the conditions for an event execution. This contract specifies the event type as well as its identifier; an optional list of affected apps, in case we want to restrict such enforcement to a subset of event dispatchers or receivers; whether the policy enforcement also takes place when the event owner (app) is in the foreground; and the rate at which they are allowed to execute. From this definition, an energy-savvy user could program her smartphone to permit calls to `WeatherUpdateService` from a weather-forecast app at most once every six hours, whereas calls to `LocationUpdateService` from the same app would remain unlimited.

To enforce user-defined policies, we outline a controller comprising three agents: observer, arbiter and actuator. The *observer* intercepts every event occurrence and bookkeeps its frequency. The *arbiter* verifies whether the measured event rate is above the policy-defined threshold, if it exists, and notifies the *actuator*, which hijacks the event continuation to artificially reduce its occurrence rate. Figure 3 illustrates our control sequence.

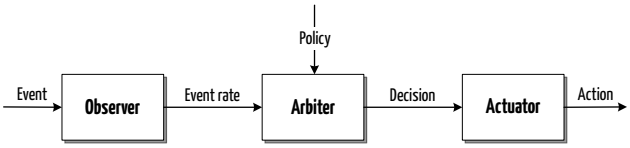


Figure 3: Sketch of our event-control system as a three-stage pipeline.

There are two ways to fulfill event throttling: canceling or delaying. An event cancel denies the continuation of its call, with an early return that prevents the payload or the event callback from running. An event delay, on the other hand, postpones the continuation of the event call for a limited time. In this work, we opt for canceling the event handling. We discuss the pros and cons of each choice in §7. It is important to understand that an event cancel

does not result in a crash. Alarms, services and broadcast receivers all run asynchronously. Wakelock requests, on the other hand, are synchronous and their denial will result in a sleeping system when a waiting task is expected to run. Still, the task is never aborted, but runs in chunks when the system wakes up. Our method prevents the triggering of unwanted events. When it is not possible, we abort the *event continuation* at the earliest opportunity to reduce the energy cost of the event payload.

To drive the implementation of our system, we establish a few requisites and considerations:

Comprehensive support for events. The control system should be inclusive. While app-specific solutions are effective, they do not scale to other programs. The stage pipeline should monitor and, if necessary, actuate on every event instance. Particulars about a specified event should be confined to its policy and not affect the controller. It is the responsibility of the policy designer to define a sane event frequency, considering, perhaps, the context and the impact of an event hijack. To implement an all-encompassing monitor system, we target an OS-level solution.

Support for power-oblivious applications. Users should not abstain from using their favorite apps even if they are power hogs. Uninstalling or suggesting alternative apps for the same purpose should not be acceptable. The system should cope with the existence of ill-behaved apps and act upon their misbehavior if directed by the policy designer.

Compatibility. Many solutions that rely on deep system introspection require extensive rewrites of system components [11, 14, 6] or even writing systems from scratch [43]. Although tempting, straying from the mainline can severely limit the userbase, especially in the case of consumer-oriented OSes. With that in mind, our solution should exhibit high compatibility and keep a minimum amount of changes to the underlying OS.

Efficiency. Mobile apps must cope with limited computational and energy resources. The control system should avoid high computational overhead to prevent high battery drain and system slowdown.

4.2 Implementation

To avoid reimplementing OS components to regulate event handling, Tamer uses the Xposed framework [42] to enable system modifications at runtime.

While requiring the device to be rooted, Xposed enables deep system modifications with no need to decompile applications nor flash the device[2]. Xposed intercepts Java methods and temporarily diverts the execution flow to function-hook callbacks for inspection and modification. Developers define these callbacks and compile them as separate modules. Function-call hooking happens by matching the method's name and signature of the declared callback with the running code. Callbacks run on the context of the intercepted application. Xposed allows for changing the parameters of a method call, modifying its return value or terminating it early. We leverage the hooking mechanism to intercept function calls originating from or directed at our events of interest. Finally, function hooks can be distributed as separate programs in self-contained APK files. They are not bound to a specific Android version and work without changes on the majority of customized Android releases, including those from Samsung, HTC, Sony, LG, and the CyanogenMod open-source community [10].

Figure 4 shows how Tamer relates to the Android OS. Tamer sits, along with Xposed, between user applications and the Java-based application framework, which serves as the foundation for the Android SDK. Events have directions, which helps us define how to write the interception payload. While service and wakelock calls originate from apps and are forwarded to the framework, alarms and broadcast receivers work in the opposite direction.

Tamer consists of a series of function hooks that interpose on the background-processing interface and act as a controller mechanism to enforce user-defined policies. To implement Tamer's event-canceling mechanism, we leverage Xposed's introspection API to explore, monitor, intercept and modify public and private classes, methods and members of the framework (Table 2). We used our knowledge on the Android SDK aligned with the source code of Android's framework stack to decide where to place the instrumentation points that would constitute our controller. We analyzed the source code stemming from each event call on the SDK's public interface. Modeling the relationships between subroutines as a call graph, we considered each interface function as a leaf node. In some occasions, we had to backtrack the call graph to find a proper instrumentation point. This was necessary for three reasons: (1) the public interface did not offer enough context to feed our monitoring system (e.g., missing receiver name, unclear caller-callee relationship, etc.); (2) in the case of receiving events, it was better to interpose on a call as early as possible to avoid unnecessary operations before a cancellation; (3) an event call may have more than one function signature, therefore we looked for a converging function node. We found one exception to the last rule when handling broadcast receivers. Applications can declare receivers in two ways: statically via a `Manifest` file or dynamically using the Android SDK API. Since the Android framework keeps separate data structures for each case, we had to instrument them separately.

Tamer's interception can suppress wakeups due to service invocations, wakelock acquires, and intent broadcasts. Because of the way Android handles alarms, our implementation can only curtail the alarm's callback payload. The system will still periodically wake up according to the alarm's schedule, but will immediately return to sleep. Our

[2] For brevity reasons, we refer readers to [41] for an explanation on how such thing is possible.

Event	Class	Method	Instrumentation Payload
Wakelock	com.android.server.PowerManagerService	acquireWakeLockInternal	Search for policy. If found, early return in case call happens before grace period. Else, let acquire proceed; bookkeep wakelock and start grace-period timer.
		releaseWakeLockInternal	Only called if there was no block; report how long wakelock was held.
Service	com.android.server.am.ActivityManagerService	startServiceLocked	Search for active policy. Proceed as in wakelock case.
BroadcastReceiver	android.app.ContextImpl	registerReceiverInternal	Add reference to API-registered receiver.
		unregisterReceiverInternal	Remove reference to API-registered receiver.
	com.android.server.pm.PackageManagerService	addActivity	Add reference to receiver registered statically.
		removeActivity	Remove reference to receiver unregistered statically.
	com.android.server.am.ActivityManagerService	broadcastIntentLocked	Search for active policy. Temporarily remove receiver from framework index to prevent event broadcasts. Update stats.
Alarm	com.android.server.AlarmManagerService	triggerAlarmsLocked	Search for active policy. Proceed as in wakelock case.
GPS (See §5.3)	android.location.LocationRequest	requestLocationUpdates	Enable GPS throttling for calling app.
		removeUpdate	Disable GPS throttling for calling app.
	com.android.server.LocationManagerService	reportLocation	Search for active policy. Let callback report incoming location, but temporarily switch off GPS sensor for blocking period.

Table 2: A brief description of TAMER's instrumentation points.

controller implementation covers all versions of Android ranging from Ice Cream Sandwich to KitKat[3]. In a handful of occasions, we resorted to different instrumentation points for a given event, mostly due to small differences in the function signatures between OS versions. Because the framework interface is fairly stable, covering future versions of Android should not require major changes.

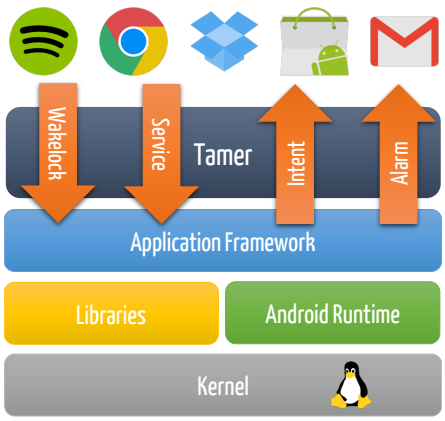


Figure 4: TAMER sits between apps and the framework stack and interposes on events between these two. Lower system layers are oblivious to our system.

5 Evaluation

We evaluate TAMER in four ways. First, we revisit our motivating scenario (§2) and use TAMER to extend the battery life of the GMS-based installation. We then investigate how TAMER can effectively mitigate energy bugs, a system behavior that causes unexpected heavy use of energy not intrinsic to the desired functionality of an application. Next, we show how to use code injection to create specialized versions of controllers for situations that our four-event toolset cannot handle. Last, we measure the overhead caused by TAMER on performance and energy.

5.1 Dealing With Google Mobile Services

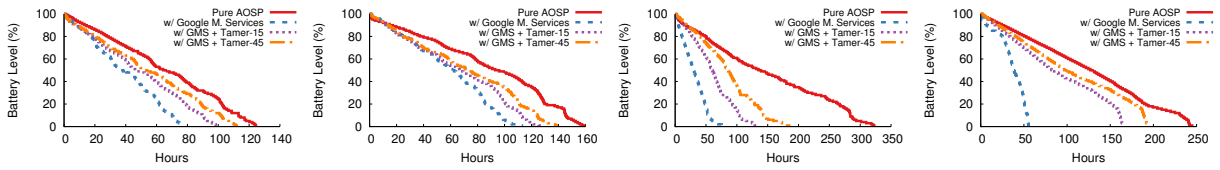
In §2, we saw how the inclusion of GMS into the baseline AOSP significantly reduced the battery life of all tested devices. Nonetheless, GMS adds a series of services and applications that truly enhance the user's mobile experience. In fact, most users do not even have the option of uninstalling them, as GMS comes pre-installed as a system package in the majority of handhelds. We show how TAMER can reach a tradeoff between GMS's functionality and battery savings. We aim to keep the added value of GMS without the cost of a silent battery depletion.

Event Name	Type	Count	Duration (s)
NlpWakelock	W	5963	1662.71
NlpCollectorWakelock	W	2121	3926.63
LocationManagerService	W	2030	67.12
NlpLocationReceiverService	S	1159	-
NetworkLocationService	S	579	-

Table 3: Top event occurrences for the Galaxy Nexus' battery drain due to GMS. A handful of events are responsible for the major impact on the battery. W signifies a wakelock event, whereas S stands for service invocation.

With the control mechanism established, our next

[3]This limitation is due to Xposed's limited OS support. Support for Android's latest release (Lollipop) is not stable enough to cover our needs.



(a) Galaxy Nexus (b) Galaxy S4 (c) Fire 2 (d) MeMO Pad 7

Figure 5: Battery drop for the four devices while idling with the screen off. After applying our policies to GMS, battery life improved in dozens of hours for all devices.

step is to design a policy that reduces the battery impact of events originating from or destined to GMS. Table 3 ranks the top triggered events reported by Tamer's monitoring module. For wakelocks, we also report the time they were held. We use event frequency as an heuristic to guide policy configuration. We note that `NlpCollectorWakelock` was primarily responsible for keeping the system awake in the background. Online reports [40, 5] indicate that `NlpCollectorWakelock` is related to `Location Reporting`, an Android feature to estimate and report the current location based on WiFi APs and cell-tower signals. Apps that use this feature include Google Now and Google Maps, among others. The other frequently reported events are also related to the same feature. Disabling location reporting on the device's Settings menu would be a logic solution to increase sleep time, if dependent apps did not stop working.

The problem with `NlpCollectorWakelock` and associated events is the frequency they wake up the system and keep it awake, which sums to a substantial period of non-sleepiness. During a discharge period of 80 hours for the Galaxy Nexus smartphone, `NlpWakelock` was called once a minute on average, whereas `NlpCollectorWakelock` contributed to keeping the system awake for more than one hour. Such a high battery impact coming from a single package does not justify the benefit of having GMS running as it is in the background. For this reason, we devised two policies for GMS, targeting `NlpCollectorWakelock` and its associated events, to alleviate this wakeup burden. For each wakelock and service in Table **??**, the first policy (Tamer-15) allows a single call every 15 minutes. The second policy (Tamer-45) allows one call every 45 minutes. Deciding on an appropriate rate is a subjective matter. Our setup tries to reach a balance between informing subordinate apps of location updates and increasing battery lifetime. Figure 5 shows that our policies substantially reduced the battery-drain rate of all tested devices.

5.2 Chasing Energy Bugs

An energy bug, or *ebug*, is a system error either in an application, OS, firmware or hardware that causes an unexpected amount of high energy consumption [35]. Such errors occur due to a variety of reasons such as programming mistakes, faulty hardware, malicious intent, etc. Because such errors may not result in a crash, users will only notice their effect when it is too late: an early dead battery.

Differently from previous research which identified and characterized energy bugs [46, 37], in our evaluation we focus on mitigating them at runtime. Tamer allows users to run the offending applications without the adverse effects of the bugs. Finding ebugs is not trivial and the lack of a centralized repository of updated samples prevents us from testing our controller more extensively. We successfully reproduced and circumvented three application ebugs described in [23]. For the other described cases, we could not confirm the existence of bugs. We assume these defects have been fixed by developer updates.

Next, we present two detailed case studies of new ebugs that we found. To identify them, we used eStar [30], a tool that ranks the contribution of apps to battery depletion. We first selected apps that display poor energy efficiency and filtered them based on high popularity at the Google Play Store (our two selections featured on the top-20 ranking of their respective categories: Games and Health & Fitness). Although eStar ranks energy-inefficient apps, we still had to manually verify whether such inefficiency was due to foreground or background activity. For each application, we simulated a user interaction consisting of a short-length active session followed by a long period in the background. **Bejeweled Blitz** [13] is an award-winning puzzle game with over 10 million installs from Google Play Store. After a 15-minute play session on the Galaxy S4 smartphone, Tamer reported that our game generated a single background event – the acquire of the `AudioIn` wakelock. Because games are resource-hungry apps, this event call initially did not instigate any suspicion. We discovered a red flag, though, after switching Bejeweled to the background: `AudioIn` was not released after the game suspension. To mitigate this bug, we wrote a simple policy targeting Bejeweled Blitz that blocks the renewal of the culprit wakelock during background time. Figure 6 depicts the battery drop of a 12-hour session with Bejeweled loaded in the background before and after applying our policy. We see a 4× improvement on battery drain. Figure 7 confirms the effect of releasing the ill-behaved wakelock: before being *tamed*, the smartphone spent approximately 95% of the time with an awake CPU. With Tamer's interposition, most of the residency ratio was converted to deep sleep.

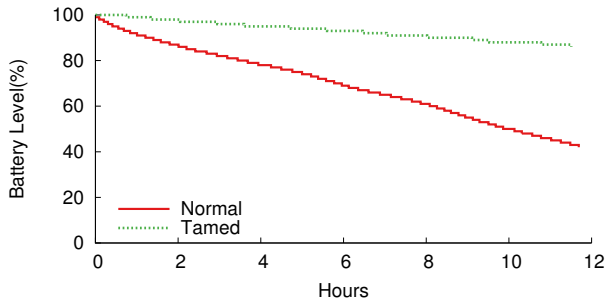


Figure 6: Battery drain over 12 hours for Bejeweled Blitz without TAMER (Normal) and with a TAMER policy blocking the `AudioIn` wakelock. In both cases, the game was started and the phone switched to idle mode with the screen off.

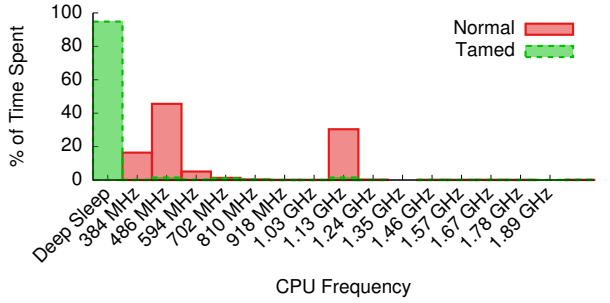


Figure 7: CPU residency time for the original and tamed versions of Bejeweled Blitz on the Galaxy S4. The tamed version spent 94% of the time in Deep Sleep.

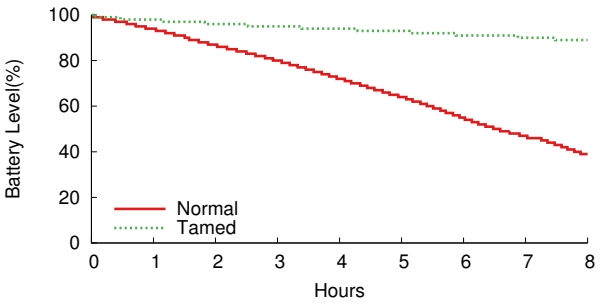


Figure 8: Battery drain over 8 hours for Nike+ Running without TAMER (Normal) and with a TAMER policy that blocked all five held wakelocks. In both curves, the app was started and the phone switched to idle mode with the screen off.

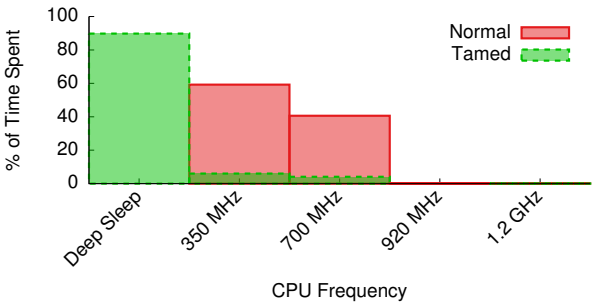


Figure 9: CPU residency time for the original and tamed versions of Nike+ Running on the Galaxy Nexus. The tamed version spent 89.4% of the time in Deep Sleep.

Nike+ Running [32] is a fitness app for tracking runs. Nike+ relies on the GPS sensor, the accelerometer, and barometer to estimate distance and speed. According to TAMER, Nike+ acquired five wakelocks while running: `AudioMix`, `FullPower Acc Sensor`, `FullPower Pressure Sensor`, `FullPower Recording` and `NlpWakeLock`. Judging from the wakelock names, we can assume a few hardware components remained awake to prevent device-sleeping. We found an ebug when pausing our running session and switching Nike+ to the background. In this case, we expected the application to release all wakelocks. Like Bejeweled Blitz, Nike+ forgot to relinquish the locks upon leaving the foreground. Our policy was also equivalent: block the culprit wakelocks during background time. Figure 8 shows the battery drop after an eight-hour session before and after applying our policy. We see a 5× improvement on battery drainage. Figure 9 displays the CPU residency on both scenarios: CPU deep-sleep residency jumped from a 0% to 89.8%. We also acknowledge a major contribution of the GPS and sensors to battery decay. Their duty cycle is equivalent to the time the homonymous wakelocks were held.

5.3 Looking at the Other Side of Events

TAMER can block events at their imminent arrival or dispatch, thus saving energy that would be consumed by their continuation. One can say that TAMER focuses on the *effect* of an event. Still, there are situations in which the energy cost of its *cause* is significant. A user could write a policy targeting an instant messenger's broadcast receiver to throttle the effect of a message arrival – a notification in the form of sound or vibration. However, she cannot block the message arrival itself, the major contributor to energy consumption in this case, as the remote sender is not covered by TAMER.

Acting on the cause of an event is complicated because its originator, when known, can take various shapes, like a disk or network I/O operation. Consequently, finding a converging instrument point in the source code that encompasses all such shapes is complex. In parallel, we should avoid point solutions that only fit one app. Between these two extremes, we can reach a balance by instrumenting code that abstracts common functionality and generates events used by a subset of applications. We consider the case of navigation apps to illustrate such scenario.

Throttling Localization. Android applications that rely on the GPS sensor follow a basic model: (1) they register a position listener and (2) they periodically receive location updates from a provider proxy, the only interface to the localization system [3]. A provider proxy serves as an interface to various location sources, including the GPS sensor, WiFi APs, and cell towers.

Given that the GPS is an energy-hungry resource, we consider a throttling mechanism for its duty cycle. Paek et al. [34] successfully demonstrated the potential energy savings of duty cycling by creating a rate-adaptive positioning system that switches the GPS sensor on and off and uses alternative location sources based on position accuracy. As a demonstration, we consider a simpler GPS-throttling implementation sans secondary sources. An advantage of our approach is the dispensing of OS recompilation, keeping it compatible with the majority of Android devices. The GPS sensor provides periodic position fixes (every second) to the OS. Some of these fixes are not relayed by the Android framework to the navigation app as they are not significantly different. We can reach a more energy-efficient navigation by directing the GPS duty cycle. We inject code into internal classes related to the framework's GPS provider and open a direct communication channel between the GPS device and our throttling mechanism. This direct channel permits our controller to switch the GPS on and off. Table 2 summarizes our instrumentation.

Evaluation. We consider two location-based apps, Google Maps and Nike+ Running. We ran these two applications separately on the Galaxy Nexus phone and applied three different throttling policies to the GPS duty cycle: location updates every one, five and fifteen seconds. The rate choice depends on the user's purpose. For pedestrian navigation, a slower update rate does not affect the estimated position as much as in the case of a highway car trip. Paek et al.'s work includes a thorough tradeoff analysis between position accuracy and energy savings. We, on the other hand, only report the potential savings. For each scenario, we programmed a pedestrian route lasting ten minutes. We turned the screen off while running the application in the background. Because of the small timespan, we compare the energy dispensed instead of battery-level drop. We used the Monsoon monitor to measure the energy consumption. Figure 10 portrays the savings per application. We observe an upper bound of 27.7% on savings for the Nike+ Running application when reducing the location-update rate to 1:15s. Although a lower update rate increases the energy savings, position accuracy is penalized.

5.4 Performance Impact

Just like a network firewall, Tamer intercepts every event-happening, inspects it, evaluates the corresponding policy criterion, and finally actuates to fulfill the policy's conditions. Because Tamer diverts the normal flow of applications, it should incur as little performance overhead and energy burden as possible. We instrumented Tamer to measure the time taken to hijack an event and perform its blockage. For the longest diverted execution flow, Tamer took, on average, 320μs to execute on the Galaxy Nexus device. With regards to energy consumption, Tamer is activated only when other applications generate events.

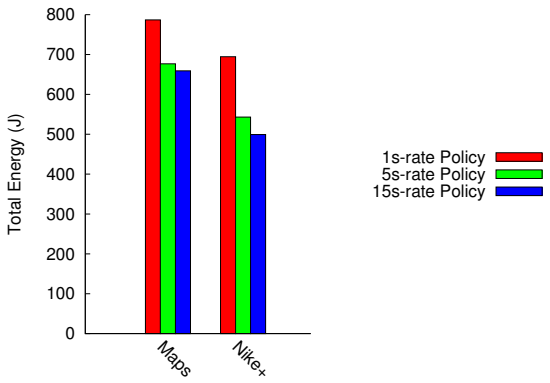


Figure 10: Total energy consumed by Google Maps and Nike+ Running after applying three different duty-cycle policies to the GPS provider. For both apps, the update rate is inversely proportional to energy savings.

Tamer does not acquire any wakelocks, but freeloads the system's active state from other wakeup sources.

6 Related Work

We are not the first to propose control of functionality in exchange for battery savings. Tamer builds upon a number of contributions to mobile power management.

Collateral Related Work. Efficient power management in mobile platforms is a challenging research problem due to the multitude of hardware configurations and power states. To improve energy consumption, we need to understand how hardware components draw power on behalf of applications. There is a myriad of tools that help quantify a device's energy expenditure. PowerScope [17] is one of the first works in the mobile domain to map energy to a program's structure. PowerScope employs linear regression and statistical sampling to apportion energy to hardware and applications. A series of recent profilers for smartphones complements PowerScope, including PowerTutor [50], ARO [38], AppScope [22], WattsOn [29] and eprof [36]. Sesame [12] and V-edge [47] go a step further and propose self-calibrating models that dispense the use of external power monitors, relying instead on internal battery data to model energy expenditure. Our analysis mainly adopts battery-drop rate as a proxy for energy consumption, but we expect such advanced contributions to be integrated by OEMs in future devices.

Saving Energy From Background Tasking. Android task killers once were the solution for background power savings, but their effectiveness is now a point of contention [27]. Task killers force background applications to quit, assuming that their removal from memory will reduce the energy footprint of released resources. Such assumption is incorrect as there is little correlation between memory and CPU usage in Android [4]. Excessive task killing may lead to the opposite effect: by discarding cached data, Android must reload apps from storage. A

killed app may restart itself immediately after being killed, using up CPU time and draining even more battery.

Rather than killing background tasks, popular apps like JuiceDefender [26] and Easy Battery Saver [1] can configure the access to power-greedy components, such as the radio and GPS, on a schedule basis. Although effective in many cases, some apps do not behave properly when they cannot, for instance, connect to the Internet. Some apps may even produce more energy overhead as they insist on accessing a resource made unavailable. Tamer circumvents such problems by mainly throttling asynchronous actions: the expecting app does not block while waiting for an event arrival or dispatch. Greenify [15] is an Android tool for hibernating apps, preventing the arrival or dispatch of events once an app switches to background. The original functionality is only restored when the app returns to the foreground. Greenify is effective in blocking misbehaving and start-at-boot apps, but its treatment of background computing is coarse and not applicable to notification-based apps that mostly run in the background (e.g., mail readers, instant messengers, calendars, etc.) Tamer is applicable in such cases as it throttles, but does not completely eliminate, background functionality.

Android also includes its own controls for background-task management. The `AutoSync` feature controls the automatic data synchronization between client apps and online accounts. With a mere tap, users can choose between disabling the synchronization of a specific feature of a selected account (e.g., photo uploads for Google+) or totally prevent any background synchronization for all registered accounts. `AutoSync` is mostly applicable to apps adopting Google Cloud Messaging (GCM), an API piece from Google Mobile Services. GCM provides a lightweight mechanism that third-party servers can use to notify mobile applications of available content to be fetched. As long as the application is subscribed to receive GCM messages, the Android device does not need to run continuously. Instant messengers, for instance, use GCM to receive notifications of new incoming messages. Tamer complements this service by offering a similar control to applications that do not adopt the GCM approach. Moreover, Tamer allows for the management of a variety of background events, whereas `AutoSync` focuses mainly on networking.

Partial inspiration for deep event monitoring stems from applications such as BetterBatteryStats [25] and Wakelock Detector [45]. Both apps report wakelock-usage statistics that developers can use to understand the root cause of battery drainage. Tamer complements such apps by empowering users to take action after they pinpoint the origin of abnormal energy consumption.

Carat [33] and eStar [30] use data collected from thousands of smartphone and tablet users to model the battery drainage of applications. By combining rich context information of multiple devices with energy awareness, it is possible to determine whether the energy used by an application deviates from its expected consumption. These tools are conservative in controlling energy expenditure, with both systems suggesting users to kill or uninstall culprit apps. eStar further recommends energy-efficient alternatives to power-hog apps, if they exist. Tamer let users keep their apps while modifying the culprit's behavior to reduce energy consumption.

7 Discussion

As any prototype, Tamer has limitations. Tamer's main utility comes from policy definition, which, at its current state, will not appeal to the end user. In the following, we elaborate on how to circumvent this usability issue. We also suggest improvements that are left as future work.

7.1 Policy Guidelines

In §5, we demonstrated how a wise policy selection can partially inhibit the surge of energy-hungry events. Our experience defining policies arose from intuition, reading source code (when available) and, in some cases, multiple attempts. At its current state, Tamer would better serve as a backend for higher-level power management tools than as an end-user app. With such limitations in mind, a user willing to run Tamer as it is, would benefit from the following guidelines to explore the event space and define effective policies.

Choosing events to control. Handhelds may carry tens or even hundreds of applications that generate thousands of events. We should not imply that policy definition must consider all of them equally. First, users prefer some apps over others. Second, event triggering does not follow a uniform distribution. As a rule of thumb, users should start with policies targeting the most frequent events. Tamer's monitoring module periodically outputs an event summary that can assist in such cases.

Cutting the red wire. Even after selecting the most prominent events for policy testing, there are no guarantees that the policy will work without side effects. Side effects may include an increase in the frequency of correlated events, the rise of unexpected events, and abnormal application behavior. Blocking alarm events recklessly could, for instance, totally defeat the purpose of a calendar app. Because most apps are only available in binary form, understanding the purpose of an event is not always clear and neither is uncovering its dependencies. Techniques used in black-box testing, such as cause-effect graphs can help. Events may also show a temporal correlation with others. To uncover temporal dependency, we generated event timelines from Tamer's monitoring output.

Use common sense. The Android OS defines two categories of applications: system and user. The former includes programs that are deemed critical, are deeply integrated into the OS, and cannot be uninstalled. Exam-

ples include the dialer, browser, and network manager, to name a few. Users apps are replaceable programs that can be freely removed and installed from the app store. As part of Tamer's design, we adopted the support of generic events. Consequently, system- and user-app events are treated equally. Policies that alter the frequency of system events may result in unwanted or abnormal behavior. Users should be mindful when defining policies involving critical events to avoid such situations. Removing support for system events would prevent such unfortunate occasions, but the definition of a system app is blurry. GMS, for example, comes pre-installed as a system package on many devices. Carriers also sell devices with *bloatware* installed as system apps. As demonstrated, systems apps present great opportunities for energy savings.

7.2 Potential Improvements

Event batching over cancellation. Our current implementation dismisses event continuation if there is a need for throttling. Alternatively, we could reschedule the asynchronous delivery of such events to coalesce multiple wakeups into one, saving even more energy. Although promising, event coalescing may lead to unexpected results that require deeper investigation. Some apps assume a fixed frequency of events. A pedometer may use the time difference between position fixes to estimate speed. Batching multiple fixes into one delivery may create havoc if the tracker does not discard outdated values. Nevertheless, coalescing has found its way in other domains. The Linux tickless kernel [44] reduced the precision of software timers to allow the synchronization of process wakeups, minimizing the number of CPU power-state transitions. From its Lollipop release, Android started to batch alarms that occur at reasonably similar times, turning them inexact. Xu et al.'s recent work on coalescing events to save energy in the context of email synchronization [48] is another successful example of careful event-handling for mobile devices. As long as developers do not assume guarantees on event delivery and commutativity, we believe coalescing should supersede cancellation as an energy-saving feature.

Native code support. Tamer controls applications by wrapping function calls from the Android Java API. Applications that make heavy use of native code, like games, multimedia apps and ELF libraries, could acquire wakelocks, spawn threads and perform background tasks using C/C++ code, thus bypassing our control system. Extending support to native code would require a similar effort on analyzing and instrumenting `libc` function calls.

Support for other mobile OSes. Background processing is not exclusive to Android, although handled differently by other mobile OSes. Apple's iOS 7+ regards background processing as a privilege [7]. Other than network transfers, common background tasks have limited time to completion and must respect the device's will to sleep, doing their processing in chunks after the device wakes up to handle phone calls, notifications and other interruptions. Windows Phone enforces background tasks to be lightweight by applying quotas to resources like CPU, memory, and network usage while apps are running behind the scenes [28]. Event-frequency control may not produce the same gains on Apple's and Microsoft's mobile devices given their stricter stance on deploying background tasks (mainly in the name of battery savings).

Feedback control. Tamer works as an open-loop controller, not using feedback to gauge whether the system needs more adjustments. During the design stage of this project, we discarded the closed-loop approach as it would require knowledge of application semantics as well as user perception of performance degradation. Modeling these two elements are hard problems beyond our scope.

8 Conclusion

This paper presented Tamer, an OS mechanism that interposes on task wakeups in Android and allows event handling to be monitored, filtered, and rate-limited. We demonstrated that Tamer substantially reduces the background energy use in popular Android applications. With Tamer, a device spends more time in low-power mode, which increases the battery lifetime significantly.

While this work shows Tamer's effectiveness as a mechanism, future work is needed to understand how to best construct policies that improve battery life while preserving application functionality. In future work, we will investigate techniques for determining if functionality is negatively impacted when exploring user visible elements (e.g., UI differences) between runs of an application with different policies. We will also explore which policies are most likely to have substantial battery savings in practice. With the combination of such techniques, we will strive to devise policies that improve battery life while retaining normal application functionality.

Acknowledgements

We thank Tim Nelson, Hammurabi Mendes, the anonymous reviewers, and our shepherd, Lin Zhong, for their feedback. Marcelo was funded in part by a generous gift from Intel Corporation.

References

[1] 2Easy Team. Easy Battery Saver. http://www.2easydroid.com.

[2] Android Developers. Best practices for background jobs. https://developer.android.com/training/best-background.html.

[3] Android Developers. Location strategies. https://developer.android.com/guide/topics/location/strategies.html.

[4] Android Developers. Managing your app's memory. https://developer.android.com/training/articles/memory.html.

[5] AndroidCentral.com. Google Services battery drain. http://forums.androidcentral.com/google-nexus-4/302559-google-services-battery-drain.html.

[6] Andrus, J., Dall, C., Hof, A. V. H., Laadan, O., and Nieh, J. Cells: A virtual mobile smartphone architecture. In *ACM SOSP'11*.

[7] Apple Inc. App programming guide for iOS. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html.

[8] ARM Limited. big.LITTLE technology: The future of mobile. http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[9] AT&T. Application resource optimizer (ARO). http://developer.att.com/application-resource-optimizer.

[10] CyanogenMod Community. http://www.cyanogenmod.org.

[11] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security'11*.

[12] Dong, M., and Zhong, L. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *ACM MobiSys'11*.

[13] Electronic Arts Inc. Bejeweled Blitz. http://play.google.com/store/apps/details?id=com.ea.BejeweledBlitz_na.

[14] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI'10*.

[15] Feng, O. Greenify. https://play.google.com/store/apps/details?id=com.oasisfeng.greenify.

[16] Flinn, J., and Satyanarayanan, M. Energy-aware adaptation for mobile applications. In *ACM SOSP'99*.

[17] Flinn, J., and Satyanarayanan, M. PowerScope: A tool for profiling the energy usage of mobile applications. In *IEEE WMCSA'99*.

[18] Gigaom. Google's killer Android L feature: Up to 36% more battery life thanks to Project Volta. http://gigaom.com/2014/07/02/googles-killer-android-l-feature-up-to-36-more-battery-life-thanks-to-project-volta.

[19] Google. Battery Historian. https://github.com/google/battery-historian.

[20] Google. Our mobile planet. http://think.withgoogle.com/mobileplanet/en.

[21] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., and Agarwal, Anant ad Rinard, M. Dynamic knobs for responsive power-aware computing. In *ASPLOS'11*.

[22] Jung, W., Kang, C., Yoon, C., Dongwon, K., and Cha, H. AppScope: Application energy metering framework for Android smartphone using kernel activity monitoring. In *USENIX ATC'12*.

[23] Kim, K., and Cha, H. WakeScope: Runtime wakelock anomaly management scheme for Android platform. In *ACM EMSOFT'13*.

[24] Kim, W., Gupta, M. S., Wei, G.-Y., and Brooks, D. System level analysis of fast per-core DVFS using on-chip switching regulators. In *IEEE HPCA'08*.

[25] Knispel, S. BetterBatteryStats. https://play.google.com/store/apps/details?id=com.asksven.betterbatterystats.

[26] Latedroid. JuiceDefender – battery saver. http://www.juicedefender.com.

[27] LifeHacker.com. Android task killers explained: What they do and why you shouldn't use them. http://lifehacker.com/5650894.

[28] Microsoft. Supporting your app with background tasks. https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh977056.aspx.

[29] Mittal, R., Kansal, A., and Chandra, R. Empowering developers to estimate app energy consumption. In *ACM MobiCom'12*.

[30] Mobile Enerlytics LCC. eStar: Because mobile devices are not mobile if they are plugged in. http://mobileenerlytics.com.

[31] Monsoon Solutions Inc. Power monitor. http://www.msoon.com/LabEquipment/PowerMonitor.

[32] Nike, Inc. Nike+ Running. http://play.google.com/store/apps/details?id=com.nike.plusgps.

[33] Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. Carat: Collaborative energy diagnosis for mobile devices. In *ACM SenSys'13*.

[34] Paek, J., Kim, J., and Govindan, R. Energy-efficient rate-adaptive GPS-based positioning for smartphones. In *ACM MobiSys'10*.

[35] Pathak, A., Hu, Y. C., and Zhang, M. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *ACM HotNets'11*.

[36] Pathak, A., Hu, Y. C. H., and Zhang, M. Fine grained energy accounting on smartphones with eprof. In *EuroSys'12*.

[37] Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *ACM MobiSys'12*.

[38] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Profiling resource usage for mobile applications: a cross-layer approach. In *ACM MobiSys'11*.

[39] Qualcomm. Trepn Profiler. http://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.

[40] Reddit. NlpWakeLock and NlpCollectorWakeLock discussion. https://www.reddit.com/r/Android/comments/1rvmlr/nlpwakelock_and_nlpcollectorwakelock_discussion/.

[41] rovo89. Xposed development tutorial. https://github.com/rovo89/XposedBridge/wiki/Development-tutorial.

[42] rovo89. Xposed module repository. http://repo.xposed.info.

[43] Roy, A., Rumble, S. M., Stutsman, R., Levis, P., Mazières, D., and Zeldovich, N. Energy management in mobile devices with the Cinder operating system. In *EuroSys'11*.

[44] Siddha, S., Pallipadi, V., and Van De Ven, A. Getting maximum mileage out of tickless. In *Ottawa Linux Symposium'07*.

[45] UzumApps. Wakelock Detector. https://play.google.com/store/apps/details?id=com.uzumapps.wakelockdetector.

[46] Vekris, P., Jhala, R., Lerner, S., and Agarwal, Y. Towards verifying android apps for the absence of no-sleep energy bugs. In *HotPower'12*.

[47] Xu, F., Liu, Y., Li, Q., and Zhang, Y. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *USENIX NSDI'13*.

[48] Xu, F., Liu, Y., Moscibroda, T., Chandra, R., Jin, L., Zhang, Y., and Li, Q. Optimizing background email sync on smartphones. In *ACM MobiSys'13*.

[49] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. ECOSystem: Managing energy as a first class operating system resource. In *ASPLOS'02*.

[50] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISSS'10*.