A Case for Lease-Based, Utilitarian Resource Management on Mobile Devices

Yigong Hu
hyigong1@jhu.edu
Johns Hopkins University

Suyi Liu
sliu92@jhu.edu
Johns Hopkins University

Peng Huang
huang@cs.jhu.edu
Johns Hopkins University

Abstract

Mobile apps have become indispensable in our daily lives, but many apps are not designed to be energy-aware so they may consume the constrained resources on mobile devices in a wasteful manner. Blindly throttling heavy resource usage, while helping reduce energy consumption, prohibits apps from taking advantage of the resources to do useful work. We argue that addressing this issue requires the mobile OS to continuously assess if a resource is still truly needed even *after* it is granted to an app.

This paper proposes that lease, a mechanism commonly used in distributed systems, is a well-suited abstraction in resource-constrained mobile devices to mitigate app energy misbehavior. We design a lease-based, utilitarian resource management mechanism, LeaseOS, that analyzes the *utility* of an allocated resource to an app at each lease term, and then makes lease decisions based on the utility. We implement LeaseOS on top of the latest Android OS and evaluate it with 20 real-world apps with energy bugs. LeaseOS reduces wasted power by 92% on average and significantly outperforms the state-of-the-art Android Doze and DefDroid. It also does not cause usability disruption to the evaluated apps. LeaseOS itself incurs small energy overhead.

CCS Concepts • **Computer systems organization** → **Reliability**; • **Software and its engineering** → *Operating systems*; • **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**.

Keywords Mobile apps; Operating system; Lease; Energy Efficiency

ACM Reference Format:
Yigong Hu, Suyi Liu, and Peng Huang. 2019. A Case for Lease-Based, Utilitarian Resource Management on Mobile Devices. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304057

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304057

1 Introduction

Mobile devices today offer massive programmability for third-party developers to write apps. For example, the Android 7 SDK provides 30,662 APIs. But rich interfaces do not mean mobile programming is easy. Indeed, app developers have to make careful optimizations when using constrained resources and consider device variability. With many possible events occurring due to user interaction, environment conditions, and runtime changes [38], reasoning about the code and testing the app is also hard [47, 58]. For app developers who lack training in system programming, these issues are especially challenging to deal with.

A severe type of defects developers frequently introduce in their apps is energy bugs [34, 35, 50, 55, 58] that drain battery abnormally fast. For example, wakelock is a mechanism in Android for apps to instruct the OS to keep the CPU, screen, WiFi, radio, etc., on active state. The intended usage is to acquire a wakelock before a critical operation and release it as soon as the operation is finished. In practice, apps may make an `acquire` request and forget to call `release` in some code paths or call it very late. Similarly, iOS manages the audio resource via audio sessions to apps. The Facebook iOS app had a buggy release [19] that would leak the audio sessions in some scenarios, leaving the app doing nothing but staying awake in the background draining the battery. That release also had a bug in the network handling code that would incur long CPU spins without making any progress.

While solutions exist to help remove energy bugs before release with better app testing [32, 51, 60], bug detection [44, 46, 56, 66], and libraries [30], complex energy bugs can still escape these tools. It is therefore important to design techniques to mitigate energy bugs at runtime. After all, energy bugs can cause damage at user side in part because existing mobile system is insufficient in protecting resources.

State-of-the-art runtime techniques [17, 41, 45, 52] monitor app resource usage, and kill or throttle apps if the usage exceeds a threshold. But making heavy use of a resource does *not* necessarily imply misbehavior. There are legitimate scenarios where the usage is justified, *e.g.*, for navigation or gaming. Blind throttling can break app functionalities. We

argue that the missing piece is a mobile resource management mechanism to continuously assess whether a resource is still truly useful to the app *after* the resource is granted.

This paper proposes that lease [40], a mechanism commonly used in distributed systems for managing cache consistency, is a well-suited abstraction to close this gap. In distributed caches, a lease is a contract between a server and a client that gives the holder rights to access a datum for the *term* of the lease. Within the lease term, the client's read accesses do not need the server's approval. After the term, if the lease is not renewed, *e.g.*, due to client crash or network partition, the server can safely proceed without waiting indefinitely.

Although mobile app energy misbehavior is a different problem domain, the essential abstraction of leases can be extended to mobile devices as a contract between the OS and an app about a resource (e.g., wakelock, GPS, sensor) with a condition on time. This abstraction brings two benefits. First, leases can tolerate sloppy resource usage mistakes that are common in apps (e.g., only release wakelock in `onDestroy`). A lease-backed resource by default expires at the end of a term unless it is explicitly renewed, either by the app or by the OS. In the cases where the lease term is larger than the needed duration, even if an app forgets to release the resource, the amount of wasted energy will be reduced. In comparison, a resource managed by the existing mechanism by default persists after being granted, unless it is explicitly released; this encourages superfluous holding. Second, compared to the blind one-shot throttling approach, a single lease allows a series of small *terms*, and at the end of each term, a lease decision is made based on the past behavior. This feedback loop allows lease decisions to adapt to changing app behavior, *e.g.*, from low resource utilization to high utilization. In this way, app developers are also relieved from the burden of carefully keeping track resources to avoid wasting energy.

We present, *LeaseOS*, a lease-based mobile resource management mechanism to mitigate app energy misbehavior. In LeaseOS, a resource granted to an app can be backed by a lease with one or multiple terms. If the app still holds the resource at the end of a term, the lease will be either extended or deferred (temporarily expired). An attempt to use resources with an expired lease later requires approval by the lease manager.

A core challenge of LeaseOS is to make appropriate lease decisions including whether to renew a lease and the length of a term. Ideally, the decisions should effectively mitigate energy misbehavior, but should *not* deprive apps of the legitimate use of resources. A straightforward lease policy is to use the resource holding time [45, 52]. But through studying real-world apps (§2), we find this metric is a misleading classifier for energy misbehavior. LeaseOS instead takes a utilitarian approach in its lease management decisions. We introduce a novel measure, *utility*, to enhance leases, which describes the quantity of "usefulness" that an app obtains from a granted resource. Thinking from the utility perspective allows us to further break down energy misbehavior into four classes and accurately capture three of them.

To transparently integrate leases into existing mobile systems, LeaseOS designs app-oblivious lease management that handles all lease operations including creation and renewal happen behind the scene. At each lease term, LeaseOS collects a set of generic utility scores to measure the resource utility information. In this way, no changes to app source code are required. LeaseOS also provides a simple API to leverage semantic information from developers by allowing apps to optionally define a custom utility function. The return value of this function is taken as a hint when the generic utility score is not too low to prevent developers from abusing this API.

We implemented LeaseOS on Android release 7.1.2. To evaluate LeaseOS, we reproduced 20 *real-world* apps with different kinds of energy defects. When running these buggy apps on LeaseOS, the wasteful power consumption can be reduced by an average of 92%. In comparison, two state-of-the-art runtime solutions, Android Doze [28] and DefDroid [45], only reduce the power consumption by an average of 69% and 62%, respectively. LeaseOS also did not cause usability disruption to the evaluated apps because the temporarily revoked resources indeed did not contribute to the utility of these apps. LeaseOS incurs <1% power overhead.

This papers makes the following contributions:

- analysis of energy misbehavior runtime characteristics and classification of misbehavior into four classes.
- a study on 109 real-world energy defect cases to understand the prevalence of different misbehavior types.
- adapting the lease abstraction into mobile system to mitigate common energy misbehavior.
- design and implementation of a lease-based, utilitarian resource management mechanism, LeaseOS.
- an evaluation that demonstrates the practical benefits of LeaseOS's approach.

2 Understanding Energy Misbehavior

To improve system-level runtime solutions for app energy misbehavior, it is useful to first understand how buggy apps behave at runtime. For this purpose, we reproduced and analyzed five real-world buggy apps on multiple smartphones and environments. We highlight three cases and our main insights in this Section, followed by a quantitative study on 109 real-world energy misbehavior cases.

2.1 Real-world Cases and Experiment Setup

Case I. K-9 mail [26] is a widely-used Android email app. The app had a defect that when the network is disconnected or the mail server fails, the app would encounter an exception and handle it by retrying indefinitely. For each retrying, the app would acquire a wakelock, which keeps the device on

causing severe battery drain. Developers fixed the issue by adding an exponential back-off and prompt wakelock release.

Case II. Kontalk [27] is a popular messaging app. When a user logs in, Kontalk authenticates with a server and establishes a connection. In one version, the app acquires a wakelock when the service is created and only releases it when the service is destroyed. This forces the CPU to stay active for a long time. Developers fixed the defect by releasing the wakelock as soon as the app is authenticated.

Case III. BetterWeather [23] is a widget that shows the weather condition based on the user's current location. One of its releases causes high battery drain when a device is unable to get a GPS lock, *e.g.*, inside a building. The root cause is that the app's `requestLocation` method keeps searching for GPS non-stop in an environment with poor GPS signals.

To capture the runtime characteristics of the buggy apps, we build a profiling tool that samples a vector of per-app metrics every 60*s*, *e.g.*, wakelock time, CPU usage (`sysTime` + `userTime`). The experiments are run on five different Android phones: Google Pixel XL, Nexus 6, Nexus 4, Samsung Galaxy S4, and Motorola G. They represent high-end to low-end smartphones with decreasing hardware capability and battery capacity. Their software ecosystems are also different, with Pixel, Samsung and Moto phones being heavily used and the Nexus phones being lightly used.

2.2 Analytical Model: *Ask-Use-Release*

Through the real-world apps, we seek answers for several questions: what is energy misbehavior? what are the common patterns of energy misbehavior? why is existing mobile resource management mechanism insufficient?

From studying the code patterns of energy bug cases, we find that the intrinsic characteristic of energy misbehavior is captured by an abstract model used in the existing mobile OSes: *ask-use-release*. Under this model, an app 1) asks for (tries to acquire) a resource; 2) after basic checks, the resource is granted; 3) the app uses the resource to do some work; 4) the app releases the resource in the end. This model, while intuitive to understand, is not friendly to use for app developers who are inexperienced in efficiently managing resources, because it makes three assumptions: *(a)* the requester can get the requested resource in a short time; *(b)* the duration of holding the resource is finite; *(c)* the resource is released as soon as the necessary work is completed. All of the three assumptions can be error-prone that lead to energy misbehavior, as analyzed in our experiments.

2.3 Experimental Results and Observations

Misbehavior in the *Ask* stage We run the buggy BetterWeather app (case III) for more than 1 hour on the Nexus phone in a building with weak GPS signals. Figure 1 shows the GPS request duration over time. Each point is measured

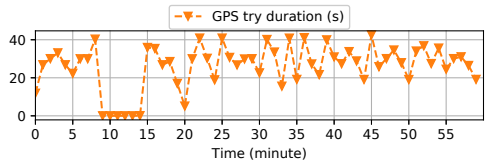

Figure 1. BetterWeather's GPS try duration every 60s.

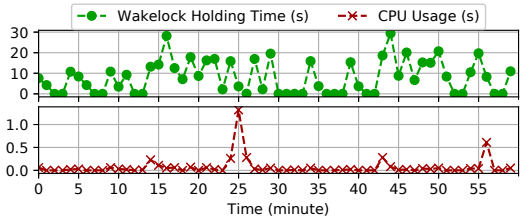

Figure 2. Wakelock holding time and CPU usage of buggy K-9 mail in a **connected environment** with a bad mail server.

at ever 60s. We can see that for each measurement interval, the app spends around 60% of the time asking for the GPS lock. But the app never gets the GPS information; all the data points in the figure, which incur significant power consumption, are spent on requesting without entering the phase of using GPS locations. Without the GPS information, the app could not obtain weather, update UI, etc. Therefore, the excessive power consumption spent in the asking stage creates almost no value to the app, thus frustrating users.

Misbehavior in the *Use* stage. After a resource is granted, an app may hold it for a long time. We run the buggy K-9 mail (case I) on the Motorola and Nexus phone in a network-connected environment with a problematic mail server. Figure 2 shows that in the majority of the one-minute measurement intervals, the wakelock holding time is long. But comparing the Motorola's measurements with the Nexus's (not shown due to space constraint), the absolute holding time and frequency of abnormal intervals differ by 2×, because of the variance in the ecosystems and hardware. In addition, several normal apps in the test phones (e.g., Pandora, Transdroid, Flym) also incur long wakelock holding time.

Therefore, a long absolute holding time for a resource could be merely an artifact of variations in different mobile systems or legitimate heavy resource usage. Using it as a classifier can flag a normal app as misbehaving. Real energy misbehavior happens when an app holds a resource for long but does not actively *utilize* the resource. For the wakelock resource (which instructs the CPU to stay active), it implies that the CPU usage would be lower than the wakelock holding period. Figure 2 shows that in the buggy K-9 mail case, CPU usage is much smaller (mostly 0) than the long wakelock hold time. This ultralow utilization (< 1%) pattern is

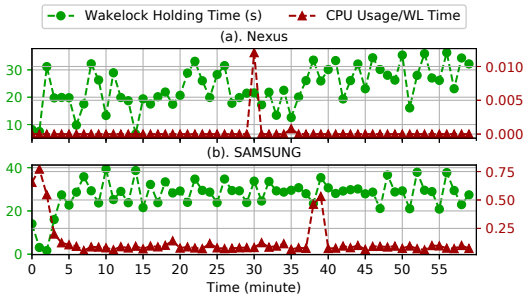

Figure 3. Wakelock holding time and ratio of CPU usage to wakelock time for the buggy Kontalk app on two phones.

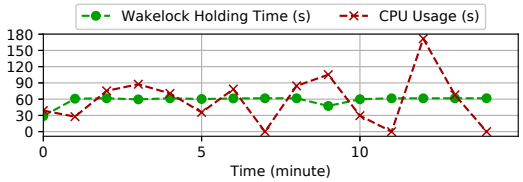

Figure 4. Wakelock holding time and CPU usage of buggy K-9 mail in a **network-disconnected** environment.

consistent across different phones and ecosystems in our experiments. Figure 3 plots the measurements from the buggy Kontalk app (case II) on Nexus and Samsung phones, which show a similar pattern. The ultralow utilization also does not exhibit in the tested heavily-used, normal apps.

Utilizing a resource well implies more than just the resource utilization ratio. A high utilization ratio does not preclude an app from misbehaving if most of the utilized resources are spent doing work useless to the users. The buggy K-9 mail (case I) has a second triggering condition that occurs in handling exceptions due to network disconnectivity. Figure 4 shows the results of running K-9 on Pixel XL under this condition. We can see that, compared to the results in a connected environment with a problematic mail server (Figure 2), the wakelock time is on average 4 times higher. The ratio of CPU usage over wakelock time is also much higher, even exceeding 100%. From utilization point of view, the results indicate that the CPU awake time K-9 mail requests is highly utilized. But from the runtime logs of K-9 mail, the app is stuck in a loop of an exceptional state doing wakelock acquisition, network request, and error handling, without making any progress. Therefore, utilizing a resource should broadly represent the *utility* of a granted resource, *i.e.*, the values to users brought by the consumed resource.

Misbehavior in the *Release* stage. When a resource is highly utilized and produces high utility, if the resource is not released after a long period or is frequently re-acquired, it can incur significant energy consumption. However, users may *not* consider this as an abnormal battery drain. For example, a user may play Angry Birds or use Facebook extensively. The shortened battery life is an expected outcome so it might be undesirable to sacrifice functionality for energy savings.

Table 1. Four types of energy misbehavior. ✓(✗) means the behavior can (not) occur for this resource. ✓*: the behavior has a different semantic for this resource.

Resource	Ask	Use		Release	
	FAB	*LHB*	*LUB*	*EUB*	*Normal*
CPU, Screen	✗	✓	✓	✓	✓
Wi-Fi radio, Audio	✓	✓*	✓	✓	✓
GPS	✗	✓*	✓	✓	✓
Sensors, Bluetooth	✗	✓*	✓	✓	✓

2.4 Energy Misbehavior Classification

Based on the experimental analysis, Table 1 summarizes four types of energy misbehavior in the *ask-use-release* model. The first three are due to clear app defects: *Frequent-Ask-Behavior* (FAB), in which the app frequently tries to acquire the resource **but rarely gets it** (*e.g.*, Figure 1), *Long-Holding-Behavior* (LHB), in which the app is granted with the resource and holds it for a long time **but rarely uses the resource** (*e.g.*, Figure 2), and *Low-Utility-Behavior* (LUB), in which the app uses the granted resource for a long time to do a lot of work **but most of work is useless** (*e.g.*, Figure 4). The fourth type is *Excessive-Use-Behavior* (EUB), in which the app does a lot of useful work **but incurs high overhead**.

Table 1 shows how the behavior types apply to different mobile resources. Not all resources can incur FAB. For example, the request to wakelock or sensors can almost immediately succeed. For LHB, the GPS and sensor resources have slightly different semantic compared with CPU due to different mechanisms. For CPU, after an app acquires the wakelock, it can do anything including not using it. But for GPS or sensors, an app registers a listener with the OS to receive location updates. When the resource is acquired, the listener is invoked. Thus the ratio of the time to collect location information over GPS holding time is almost always 100%. We define the LHB for GPS/sensor as the utilization of the GPS location *data* rather than the physical resource.

Our observations from the experiments suggest that the unique characteristic of energy misbeheavior across different apps and ecosystems centers around how well a resource is utilized. In particular, when energy defects are triggered, the resource request success ratio ($\frac{\text{unsuccessful request time}}{\text{total request time}}$) or utilization ratio ($\frac{\text{resource usage time}}{\text{holding time}}$) or utility rate ($\frac{\text{utility score}}{\text{resource usage time}}$) quickly drops to a very low value in a short time and stays low for a relatively long period. The three metrics respectively identify FAB, LHB and LUB. The quick-drop observation also implies that, to catch energy misbehavior early on,

Table 2. Prevalence of each type of energy misbehavior in 109 real-world cases. The unknown (N/A) cases are because the app is closed-source or the issue is not resolved yet.

Type	Number of cases					Pct.
	Bug	Config.	Enhance.	N/A	Total	
FAB	10	1	1	0	12	11%
LHB	18	5	0	0	23	21%
LUB	23	4	1	0	28	26%
EUB	8	18	5	3	34	31%
N/A	0	0	0	12	12	11%

we do not need to sub-divide the lease term into very small epochs and check each epoch. Checking the resource utility metrics at the end of a lease term is sufficient.

2.5 Prevalence of Misbehavior Types

To understand how our observations generalize beyond the few cases we analyzed in previous Section, we conduct a study on 109 real-world energy misbehavior cases in 81 popular apps. All the apps and issues are collected from open source hosting service [24, 25] or popular user forums [22, 31]. We study the root cause of each case based on the comments and source code. To understand the severity of each case, we classify the root causes into bug, configuration/policy and enhancement. Bug means the energy waste is caused by a software defect; configuration means developers made an intentional choice to trade energy efficiency for other properties (*e.g.*, accuracy); Enhancement means some optimization that developers could add. Bug usually has high severity and priority for developers to fix. Table 2 shows the distribution of different energy misbehavior types.

Finding 1: *All four types of misbehavior are prevalent. FAB, LHB and LUB together occupy 58% of the studied cases while EUB occupies 31% of the cases.*

Finding 2: *The majority (80%) of FAB, LHB and LUB due to clear programming mistakes (Bug), while the majority (77%) of EUB are due to design trade-off (non-Bug).*

Based on these two findings, we believe that the EUB is the grey area between normal behavior and misbehavior, and the first three classes should be the primary target for a runtime mitigation mechanism to be acceptable to users.

3 Lease Abstraction for Mobile System

3.1 Abstraction

In existing mobile OSes, when an app requests a resource, the OS performs an initial sanity check; if the check passes, the app's right to hold the resource persists indefinitely unless the app explicitly renounces it. This *ask-use-release* model assumes that apps are capable of efficiently managing resources throughout the resource lifetime, which is problematic.

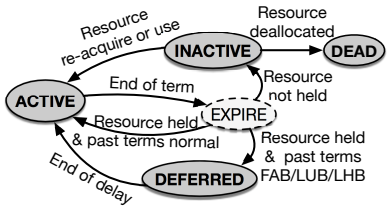

Figure 5. Lease state transition.

LeaseOS explicitly manages the resource rights by introducing leases. A lease essentially grants an app the rights to request and use a specific resource instance (in the form of a kernel object). This right is honored for a period of time, the lease *term*. In the current mobile OS scheme, an app accesses the kernel object through a bound wrapper in the app address space. With LeaseOS, a lease is created when an app first accesses the kernel object, and is destroyed when the corresponding kernel object is dead. A lease can last for multiple terms, $t_1, \ldots, t_n$. An app can hold multiple leases at the same time, each uniquely identifiable with a lease descriptor.

During a lease term, t_i, the lease holder possesses the right to access the resource instance and does not require approvals from the OS. At the end of t_i, the OS decides whether to renew (extend) the lease. In other words, a lease in LeaseOS represents a timed capability. A lease term can range from zero to infinity. A zero-length term means every access needs to be checked by the OS. A lease with infinity term means the OS will not do any check after the resource is granted to the app, which essentially degrades to the existing *ask-use-release* model.

3.2 Lease States

In distributed systems, a lease has two simple states: the *active* state when the lease is created or renewed, and the *expired* state when the term ends. We adapt leases for efficient mobile resource management. The resulted lease states and transitions are slightly more complex (Figure 5). When a lease term ends, if the app still holds the resource, the resource utility metrics (Section 2.4) in the past term are checked. For normal behavior, the lease will be immediately extended with a new term and switch to the active state again. LeaseOS introduces a new *deferred* state. If the behavior is one of the three misbehavior (FAB, LHB, LUB), the lease enters the deferred state in which the lease will be extended but with a delay interval τ. During τ, the capability and resource associated with this lease is *temporarily* revoked to reduce wasteful energy consumption. After τ, the capability and resource is restored and the lease transits to active state again. The deferred state essentially is a controller to slow down low-utility app executions (Section 4.6)

If, when the lease term is expired, the resource is no longer held, *i.e.*, the app calls resource release at some point in the term, the lease transits to the *inactive* state. When the lease is inactive, if the app tries to re-acquire or use the resource

```
public class ServiceRotationControlService {
+  ClickUtility utility = new ClickUtility();
+  class ClickUtility implements UtilityCounter {
+    public List<OrientationButtonOverlay.Event> events;
+
+    @Override
+    public float getScore() {
+      if (events == null || events.size() == 0)
+        return 50.0;
+      int click = 0;
+      int rotation = 0;
+      for (OrientationButtonOverlay.Event e : events) {
+        rotation++;
+        if (e.click == true)
+          click++;
+      }
+      return 100.0 * click / rotation;
+    }
+  };

  @Override
  public void onCreate() {
    sensor = new PhysicalOrientationSensor();
-   sensor.enable();
+   utility.events = orientBtnOverlay.mEventList;
+   sensor.enable(utility);
  }
}
```

Figure 6. Custom utility counter in app TapAndTurn.

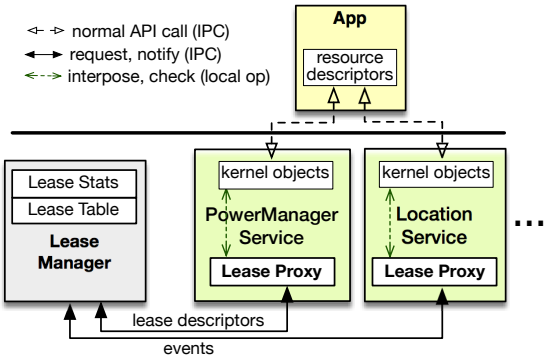

Figure 7. Architecture of LeaseOS.

with an expired lease, the access requires a check with the OS who will make a renewal decision. When the lease-backed resource is fully deallocated, the lease enters the *dead* state. A dead lease can no longer be renewed and will be cleaned.

By regularly examining each term, and revoking under-utilized resources temporarily for τ, the energy waste is reduced without significantly impairing app utilities. Moreover, when an app only under-utilizes resource for a limited period, and can later efficiently use the resource, the app has a chance of getting the lease renewed and returning to normal behavior. This continuous *examine-renew* model differs LeaseOS from other simple *one-shot* throttling solutions.

3.3 Lease and Utility Metrics

In order to make lease decisions based on how useful an allocated resource is to the app, we enhance the lease abstraction with lease stat. Each lease term records a stat that contains the three broad utility metrics (Section 2.4) to classify the resource use behavior during that term. We describe the utility metrics for wakelock, GPS and sensor as an example.

Frequent-Ask occurs when an app frequently tries to acquire the resource but rarely gets it. This could occur for GPS in an environment with poor signals, but not for wakelock or sensor. The metrics include the total request time and the request duration with failed GPS lock. If the request is frequent or long but the success ratio is lower than a threshold, an FAB arises. *Long-Holding* happens when the app is granted with the resource and holds it for long but rarely uses it. For wakelock, the ratio of CPU over wakelock holding time represents the utilization. For GPS and sensor, because the app-supplied listener is always invoked, the ratio of the lifetime of the app Activity *bound* to the listener over the lifetime of the listener is a more appropriate utilization metric in this case. *Low-Utility* behavior refers to the utilization rate of a granted resource is high but most of the work is of little value. To quantify the usefulness, we use a utility score of 0 to 100. If the score is less than a threshold, a LUB occurs. While utility score is often app-specific, it is possible to use some conservative heuristics to determine a generic utility. In particular, we use the frequency of severe exceptions raised in apps for the low utility of wakelock, the distance moved for the utility of GPS, and the UI updates and user interactions with the apps as high utility.

In addition to generic utility, LeaseOS provides an *optional* simple callback interface, `IUtilityCounter`, for apps to provide custom utility. Figure 6 shows how a screen control app, TapAndTurn, can implement the interface. The app provides an icon for users to control screen rotation. If the orientation sensor detects a change in direction, an icon will appear on the screen for users to click. The developer may record the number of times that the icon occurs and the number of clicks on that icon. Then she can implement `IUtilityCounter` by returning the ratio of the clicks over the icon occurrences, multiplied by 100. For a fitness tracking app, the custom utility function could be the amount of tracking data written to the database in a period, normalized to 0–100. The custom utility is only taken as a hint when the generic utility is not too low to prevent abuse of this function.

4 Design of LeaseOS

LeaseOS is a runtime solution to mitigate energy defects in mobile apps. Specifically, LeaseOS targets addressing the *Frequent-Ask*, *Long-Holding*, and *Low-Utility* misbehavior in the *ask-use-release* model (Section 2.4). Addressing *Excessive-Use* is a non-goal. We make this design decision because FAB, LUB or LHB is clear energy misbehavior due to programming mistakes. EUB, on the other hand, is caused by heavy use of resources, which is often an intentional trade-off and is controversial to judge as misbehavior.

Table 3. LeaseOS interfaces for lease proxies. `setUtility` is exposed to apps.

Interface	Description
`long create(in ResourceType rtype, int uid)`	create a lease for a resource for app with uid
`bool check(long leaseId)`	check whether the lease is active or not
`bool renew(long leaseId)`	renew the lease
`bool remove(long leaseId)`	remove the lease
`void noteEvent(long leaseId, in LeaseEvent event)`	report an event about a resource backed by lease with id leaseId
`void setUtility(int type, in IUtilityCounter counter)`	register a custom utility function to be referenced
`bool registerProxy(int type, ILeaseProxy proxy);`	register a lease proxy with the lease manager
`bool unregisterProxy(ILeaseProxy proxy);`	unregister a lease proxy with the lease manager

4.1 Overview

Figure 7 shows the architecture of LeaseOS. A system component, Lease Manager, manages all the leases in the system. The lease manager handles lease related operations such as creation and transition of lease states. In order to make lease decisions, the lease manager also keeps track of key lease stat measuring the utility of resources associated with a lease. Since mobile apps are latency-sensitive, the lease management operations should avoid incurring excessive overheads. For this purpose, LeaseOS designs a few light-weight *lease proxies*. Each lease proxy manages one type of constrained mobile resource, e.g., wakelock, GPS. The proxy is placed inside the OS subsystem managing that type of resource and interacts with the lease manager on behalf of the apps.

4.2 Achieving Transparent Lease Integration

Under current mobile OSes, apps and OS subsystems live in different address spaces. When an app successfully gets a resource (e.g., wakelock) from the subsystem managing that resource, it obtains a wrapper on a resource descriptor, which can be used locally in the app's address space. In Android, the resource descriptor is usually a unique client IPC token, an `IBinder` object. The wrapping is provided by the system package, e.g., `android.os.PowerManager`. The real resource is a kernel object in that subsystem, e.g., an `IBinder` IPC token that is associated with the app's token. Since the app resource descriptor and the kernel object has a one-to-one mapping, invoking a resource operation on the descriptor translates to making an IPC to the OS subsystem, which manipulates the corresponding kernel object.

With LeaseOS, apps still make resource requests to the subsystems via IPC and the resource descriptors as usual. A lease proxy transparently makes lease requests on behalf of the apps to the lease manager. It maintains a mapping between the kernel object corresponding to the app resource and the lease descriptor returned by the lease manager. Because a lease proxy lives in the same address space as the subsystem, it can directly manipulate the kernel object to apply operations on resources as instructed by the lease manager, without going through or manipulating the resource descriptors in the app address space. In this way, leases are seamlessly integrated into the systems without rewriting apps. Therefore, LeaseOS is compatible with existing apps.

4.3 Lease Manager

Lease manager creates, expires, renews and removes leases for resources. When an app is granted with access to a resource instance for the first time, a lease is created. The lease is assigned with a unique lease descriptor and an initial term. The app is recorded as the lease holder. The lease manager maintains a table that contains all the leases created in the entire system for different resources granted to all apps.

For each lease term assigned, the lease manager schedules a check after the term expires. LeaseOS makes lease decisions based on app resource usage behavior. At the end of each lease term, the lease manager calculates the resource success ratio, utilization and utility stats. With the resource stats, the lease manager judges the type of resource usage behavior (Section 2.4) in the past lease term. For each lease, a bounded history of the stats and behavior types for the past terms is kept in the lease manager. Given the behavior types for the current term and last few terms, the lease manager makes a decision to renew or temporarily expire a lease.

Lease manager provides a set of APIs to the lease proxies. Table 3 shows the interfaces. A proxy calls `registerProxy` to register with lease manager to enable lease management for a type of resource. Lease proxies invoke `noteEvent` to notify lease manager about important event about the kernel object, e.g., the app calls `release` on the kernel object or the app attempts to re-acquire the object. These events will be analyzed at the end of a term to calculate stat like the resource holding time. In LeaseOS, lease expiration and renewal decisions are made by the lease manager. Lease proxies provide an `onExpire` and `onRenew` callback to be invoked by the lease manager. When a lease proxy detects an app attempting to use a resource with an expired lease, the proxy will invoke `renew` API to request lease extension. When the leaseholder (an app) dies, system services from which the holder have requested resources will clean up the kernel objects. Under

this condition, the lease proxies also need to notify the lease manager to clean up all the related leases by invoking `remove`.

4.4 Lease Proxy

Lease proxies are light-weight delegates of the lease manager. They directly interpose and check the resources (kernel objects) backed by leases. For each lease created by the manager, the lease proxy stores the mapping between the kernel object and the lease descriptor so that when the manager makes decisions about a lease, the proxy knows which kernel object to apply the operation. The proxies do not store or manage the lease content or stats. They cache the state for a lease for efficient checking. Lease proxies communicate with lease manager using lease descriptors. The communication between a lease proxy and lease manager is bi-directional. When a lease proxy starts, it will register with the manager, create and keep an IPC channel for communication.

If an app makes certain resource request operations to the proxy's host subsystem, the proxy may invoke an API of the lease manager such as `create` via the IPC channel. In addition, the lease proxy will provide several required callbacks to the lease manager, such as `onExpire` and `onRenew`. When a lease is expired or renewed, the lease manager will invoke these registered callbacks. Within these callbacks, the lease proxy will update its local lease descriptor table, the cached lease state and the state of its host subsystem to reflect the change. For instance, when an app invokes `acquire` on a wakelock instance, the power manager subsystem essentially adds the kernel object, `IBinder`, into an internal array, which will be checked to determine if the CPU should enter deep sleep mode. In this case, the lease proxy in the power manager needs to remove the `IBinder` from the array inside `onExpire`.

4.5 Lease Mechanism from Apps' Perspective

With lease proxies, enabling lease-based resource management does not require any app code changes unless apps choose to implement the optional custom utility function. Figure 8 uses a real-world app (K-9 mail) as an example to show the lease mechanism from the apps' perspective.

❶ creates a unique resource descriptor `wkLock`, with the power manager OS subsystem creating a corresponding unique kernel object (not shown). When ❶ is executed for the first time, a new lease is created behind the scene by the proxy. Before ❹ is reached, multiple lease terms may have passed. In the normal scenario, in each lease term the app does some useful work with the resource, so the lease will be immediately renewed when the term expires. When the lease term containing ❹ finishes, the lease manager finds that the wakelock resource is no longer held, so the lease transits to the inactive state. Some time later, function `start` is executed again. Upon ❶, the lease capability immediately goes back to active. When the pushing service is eventually stopped, the lease proxy death recipient immediately requests the manager to remove the lease. During this process, even if

```
public class EasPusher {
❶ TracingWakeLock wkLock = pm.newWakeLock();
  public void start() { // periodically executed
❶   wkLock.acquire();
    while (!stop) {
      try {
        Serializer s = new Serializer();
        for (String folderName : folderNames)
❷         s.start(folderName);
        s.end().done();
❸       HttpResponse resp = sendHttpClientPost();
        ...
      } catch (Exception e) {
      }
    }
❹   wkLock.release();
  }
}
```

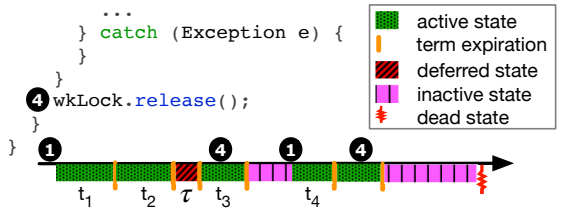

Figure 8. Lease mechanism from an app's perspective.

the absolute holding time between ❶ and ❹ is long, the app will behave the same as without lease mechanism.

The app may exhibit energy misbehavior in some lease term. For example, when the network is disconnected, it will get stuck in an exception loop due to ❸ while holding the wakelock for a long time. In this scenario, the lease mechanism will apply a penalty to the app by deferring the renewal of the next lease term for a penalty period of τ. During τ, the kernel resource is temporarily revoked to reduce wasted energy but is restored after τ. If the network re-connects, the energy misbehavior will be gone. So are the lease terms renewed again. Compared to the one-shot throttling, the lease mechanism can adapt to temporary energy misbehavior.

4.6 Implication of Deferring Lease Term Renewal

The semantic of the lease deferred state is that the capability and resource is *temporarily* revoked for τ. This revocation by mutating the state of the OS subsystem with the kernel object. But the resource descriptor in the app address space is still valid to be used by apps to make IPCs during τ. The app logic is not affected either. For `acquire` IPC, the OS subsystem essentially *pretends* it succeeds. For `release` IPC, the event will be recorded by the proxy. If no `release` occurs during τ, the temporarily revoked resource will be restored after τ.

The main penalty incurred to apps is that the low-utility execution may be slowed down. Take Figure 8 as an example. Suppose when the code execution reaches ❷, the lease for `wkLock` enters the deferred state. The lease proxy will remove the `IBinder` object from an internal array in the power manager service, without modifying the `wkLock` descriptor. If this happens to be the last `IBinder` object in the array, the phone enters deep sleep mode. So the execution is paused and will be resumed seamlessly later. If the paused execution involves network operation (e.g., ❸), when the execution resumes, an I/O exception due to timeout might occur. But the app is already required to handle such exception at compile time.

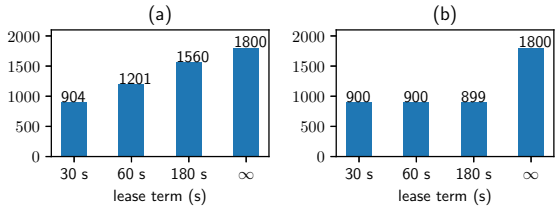

Figure 9. Resource holding times (s) of a test buggy app with Long-Holding misbehavior under different lease terms.

Therefore, the lease deferral does not cause unknown exceptions to apps. For listener-based resources like GPS and sensor, the deferred state means the app listener callback will not be invoked (or less frequently). Slowing down low-utility execution usually does not cause undesirable impact to users because it produces little value anyway (e.g., non-stop retry of failed ❸).

5 Lease Policy

5.1 Choosing the Lease Term and Deferral Interval

For a lease-based mechanism, the choice of the lease term is important. In the original distributed cache scenario, the lease term affects the trade-offs between lease renewal overhead and the extra delays due to false sharing. In our case, the false sharing trade-off does not exist because most energy-consuming mobile resources are shared in a subscription style that does not require coordination. But lease term, t, as well as the deferral interval, τ, that we introduced, influence the effectiveness on mitigating energy misbehavior and impact to legitimate resource usage. A short lease term allows the lease manager to quickly detect energy misbehavior. But it can incur high lease accounting overhead. A short deferral interval can reduce the cost of misjudgment (slowdown of legitimate high-utility executions). But it has limited effect on reducing wasteful energy consumption.

We analyze how lease term impacts the effectiveness of mitigating misbehavior. Suppose that an app starts to exhibit the *Long-Holding* misbehavior at the beginning of ith lease term, and at the end of the jth lease term, LeaseOS detects the Long-Holding pattern. The resource holding time, $H = n \times t$, and total time, $T = (n \times t) + \tau$, where $n = j - i + 1$. So the reduction ratio of wasted energy consumption, r,

$$r = \frac{H}{T} = \frac{n \times t}{(n \times t) + \tau} = \frac{1}{1 + \lambda}, \text{where } \lambda = \frac{\tau}{n \times t} = \frac{avg(\tau)}{\text{lease term}}$$

This implies that if an app holds non-utilized resource for a time longer than that the lease term, then for the Long-Holding misbehavior, the larger λ is, the more effective lease mechanisms can help reduce wasteful energy consumption. In addition, the absolute lease term is not the deciding factor. The ratio it has with the average deferral interval is the key.

To validate the above analysis, we wrote a test app that simulates the Long-Holding misbehavior based on a real-world buggy app (Torch). The test app acquires a wakelock and holds the wakelock for 30 minutes without doing anything and never releases it. For experiment purpose, the lease term is set to be 30s, 1min, 3min and ∞ (no-lease). The deferral interval is set to be 30s, which means the λ is 1, 0.5, 1/6, respectively. Each experiment is run for 30 minutes. During the experiment period, the lease states alternate between ACTIVE and DEFERRED state (i.e., $n = 1$). Figure 9 (a) shows the result. We can see that if the lease term is 30s, the app only holds wakelock for about 15 minutes, which is about half of the holding time without lease mechanisms. When the lease term increases to 1min, the holding time increases to 20 minutes and when the lease term is 3min, the holding time is about 26 minutes. We repeat the experiment with the same three terms but keep λ 1. Figure 9 (b) shows the result. We can see that the wakelock holding time under different lease terms are almost the same. This confirms our analysis conclusion. The conclusion above can also be adapted for Frequent-Ask and Low-Utility misbehavior since their lease state transitions are the same. Based on our analysis and empirical experiments, LeaseOS sets the default lease term as 5 seconds and the default deferral interval as 25 seconds.

5.2 Optimizing for The Common Case

The goal of introducing lease mechanism to mobile systems is to reduce the wasteful energy consumption due to some misbehaving apps. For many users, the majority of the apps in their devices use resources in a relatively reasonable way. Even for the misbehaving app, the energy misbehavior is often intermittent (e.g., when the network connectivity or the GPS signal is weak). This creates an opportunity for LeaseOS to optimize for the common case, normal resource usage behavior, with adaptive lease terms. In particular, if an app has been using resources efficiently, LeaseOS can increase the next lease term. An increased lease term will reduce the performance overhead, as well as unnecessary deferral for transient misbehavior. Therefore, the lease manager will increase the lease term to 1 minute if the past 12 terms (1 minute) are normal, and further increase it to 5 minutes if the 120 terms are normal. It will revert to 5-second lease term if any term in the look-back window has misbehavior.

6 Implementation

We implemented LeaseOS on top of Android 7.1.2, with 9,100 lines of code changes made to the core Android framework. Much of the logic for different lease proxies are the same, such as communicating with lease manager, maintaining mappings of kernel object and lease descriptor. This common logic is provided via a generic lease proxy class. Enabling lease management for a new type of resource therefore does not require significant efforts. It is done by inheriting from

Table 4. Average latency of major lease operations in *ms*.

Create	Check (Acc)	Check (Rej)	Update
0.357	0.498	0.388	4.79

this proxy, implementing a few proxy callbacks, and inserting some hooks in the system service. Our modifications to each enhanced system service are only around 200 lines of code.

One implementation challenge is to track exceptions from apps for generic lease utility. These exceptions are handled by the Android libcore. The libcore itself cannot directly use system APIs to pass the exception information to Android framework. To address the issue, we define a new class in the libcore `ExceptionNoteHandler` with a get and set interface. During the runtime initialization of an app process within the system service, we set a global handler that will notify the lease manager service when called. When an app throws an exception, the libcore will check if the handler is set and if so the handler will be invoked.

7 Evaluation

This Section evaluates LeaseOS. We measure how effective is LeaseOS in mitigating the *Frequent-Ask*, *Long-Holding* and *Low-Utility* energy misbehavior (Section 2.4), by running buggy apps in LeaseOS and comparing the power consumption with running them in the vanilla Android. We also measure LeaseOS's usability impact and performance overhead.

7.1 Experiment Setup

The main experiments are performed on a Google Pixel XL phone running LeaseOS. The device has a 2.15GHz quad-core CPU, 32 GB storage, 4 GB RAM and a 3,450 mAh battery. To measure the effectiveness of mitigating energy misbehavior, we need to obtain app-level power consumption. This is done with the high-accuracy Qualcomm Trepn profiler [29] and the Android built-in power profiler. For measuring system-wide power consumption, we use the Monsoon hardware power monitor. Because the battery of Pixel phone is difficult to integrate with Monsoon power monitor, we use a Nexus 5X phone as a substitute to set up the measurement (Figure 10). To make sure the baseline Android system has the same app ecosystem, we provide a flag in LeaseOS to completely turn off the lease service. The initial lease term is set to be 5 seconds with the default deferral interval being 25 seconds.

7.2 Micro Benchmark

We first conduct a micro benchmark on the performance of major lease operations. We write a test app that acquires and releases different resources 20 times. Then we collect the latency for each lease operation. Table 4 shows the results. We can see that the operations are fast, close to the Android IPC latency. As a comparison, we measured the latency for an app to make a resource acquire IPC without

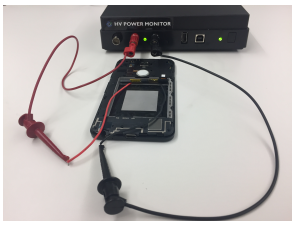

Figure 10. Nexus 5X with Monsoon power monitor.

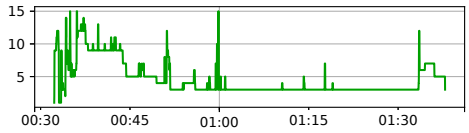

Figure 11. Number of active leases in one hour period

lease is around 2 *ms*. The lease update operation is slightly higher than creation and check because it needs to calculate the utility metrics. But lease update does *not* force pause to app execution flow. The overall latency impact to apps is small, especially since lease operations are not in the app critical paths most of the times.

We also measure the lease activities under normal usage scenario. During the experiment period, we actively use popular apps including playing games, browsing social network, reading news and listening to music for 30 minutes and then leave it untouched for another 30 minutes. Figure 11 plots the number of active leases over time. It shows the active leases are moderate and match user activities. In total, 160 leases are created. Most leases are short-lived, with a median active period of 5 seconds. But the max period is 18 minutes. The average number of lease terms are 4, and max 52.

7.3 Mitigate Energy Misbehavior

We reproduced 20 energy bug cases representing different misbehavior types in popular *real-world* apps. 5 of the cases are used in the early study (Section 2). We compare the power consumption of running these buggy apps on the vanilla Android (w/o lease) with running them under LeaseOS (w/ lease). The rise of app energy misbehavior has motivated several recent works. Starting from 6.0, Android introduces the Doze mode [28] which defers app background CPU and network activity when the device is unused for a long time. Amplify [11] and DefDroid [45] throttle excessive requests. We compare the effectiveness of LeaseOS with Doze and the simple throttling approach (with the throttling settings from DefDroid [45]). Each experiment is run for 30 minutes, during which the power consumption is sampled every 100 *ms*.

Table 5 shows the averaged power consumption under different solutions. We can see that LeaseOS can significantly reduce the wasted power consumption for all cases, achieving an average reduction ratio of 92%. The default Doze is

Table 5. Evaluate real-world apps with Frequent-Ask, Long-Holding and Low-Utility misbehavior. *: the default Doze mode is too conservative to be triggered for most cases. We made it aggressive by forcing it to take effect at each experiment.

App	Category	Res.	Behavior	Power (mW)				Reduction Percent (%)		
				w/o lease	w/ lease	Doze*	DefDroid	LeaseOS	Doze	DefDroid
Facebook [5]	social	CPU	LHB	100.62	1.93	18.92	12.68	98.08	81.19	87.40
Torch [7]	tool	CPU	LHB	81.54	1.30	19.26	14.39	98.41	76.38	82.35
Kontalk [13]	messaging	CPU	LHB	29.41	0.39	16.84	15.99	98.67	42.74	45.63
K-9 [6]	mail	CPU	LUB	890.35	81.62	195.2	136.14	90.83	78.08	84.71
ServalMesh [10]	tool	CPU	LUB	134.27	1.37	30.54	14.88	98.98	77.25	88.92
TextSecure [18]	messaging	CPU	LUB	81.62	1.198	18.78	16.78	98.53	76.99	79.44
ConnectBot [4]	tool	screen	LHB	576.52	23.23	573.23	115.56	95.97	0.57	79.96
Standup Timer [2]	productivity	screen	LHB	569.10	13.26	544.46	61.82	97.67	4.33	89.14
ConnectBot [1]	tool	Wi-Fi	LHB	17.08	0.78	3.21	2.57	95.43	81.21	84.95
BetterWeather [15]	widget	GPS	FAB	115.36	2.59	20.38	39.97	97.75	82.33	65.35
WHERE	travel	GPS	FAB	126.28	23.33	20.42	69.62	81.52	83.83	44.87
MozStumbler [16]	service	GPS	LHB	122.43	67.53	36.48	62.7	44.84	70.20	48.79
OSMTracker	navigation	GPS	LHB	121.51	8.39	20.52	73.34	93.10	83.11	39.64
GPSLogger [8]	travel	GPS	LHB	118.25	4.33	21.98	70.7	96.34	81.41	40.21
BostonBusMap [3]	travel	GPS	LHB	115.5	3.97	19.5	71.09	96.56	83.12	38.45
AIMSCID [12]	service	GPS	LUB	119.43	4.50	23.91	73.31	96.23	79.98	38.62
OpenScienceMap [14]	navigation	GPS	LUB	123.97	3.40	19.91	91.25	97.26	83.94	26.39
OpenGPSTracker [9]	travel	GPS	LUB	360.25	1.32	19.91	237.41	99.63	94.47	34.10
TapAndTurn [20]	tool	sensor	LUB	11.72	1.87	3.95	4.41	84.04	66.30	62.37
Riot [21]	messaging	sensor	LUB	19.17	1.43	6.64	3.93	92.54	65.36	79.50
							Average:	92.62	69.64	62.04

very conservative (e.g., after the phone is idle for a long time and there is no angle change in 4 minutes), as it is a system-wide mode that applies to all apps. It is triggered for only 8 cases. To evaluate whether relaxing its triggering condition can help, we force it to take effect at the beginning of each experiment through `adb` command line. Table 5 shows that even though the aggressive triggering helps, it is still much less effective than LeaseOS because any non-trivial activity can interrupt the deferral. Similarly, LeaseOS significantly outperforms the blind throttling solution. This is because the mechanism inherently cannot distinguish legitimate behavior from misbehavior so its settings have to be conservative. LeaseOS continuously analyzes an app's resource usage and utility at the end of each lease term and can take proactive action to prevent wasteful asking or holding of resources.

7.4 Usability Impact

LeaseOS's high effectiveness in mitigating app energy misbehavior does not come at a price of reduced app usability. For all of the cases we evaluated, LeaseOS did not introduce any negative usability impact. This is because LeaseOS by design optimizes for apps' utility. The three types of misbehavior we target—Frequent-Ask (but rarely gets it), Long-Holding (but do little work with it), and Low-Utility (but mostly do useless work)—all contribute little to the usability of apps. As an anecdotal user experience, the primary author has been actively using the Google Pixel phone running LeaseOS for more than 10 days and has not experienced any visible side effect or sluggish app interactions due to leases.

To further demonstrate the importance of LeaseOS's utilitarian approach, we compare how *normal* background apps perform under LeaseOS with running in a time-based throttling system (essentially leases with only a single term). We choose three representative normal background apps: 1) *RunKeeper* that tracks fitness activities with location and sensor recording in the background; 2) *Spotify* that streams music in the background; 3) *Haven* that continuously monitors intruders using sensors and cameras. For all of the three apps, LeaseOS would continuously renew leases without introducing any interruption, because the resources are utilized well. In comparison, under pure throttling scheme, all of the three apps experienced some disruption, e.g., fitness tracking, music streaming or monitoring stopped. Interestingly, we also found that the profiling tool we use, Trepn profiler, also stops collecting data, whereas it functions well under LeaseOS.

7.5 Sensitivity to Lease Policy

The effectiveness of LeaseOS depends the choice of lease term and deferral interval. For single misbehavior, Section 5.1 analyzes the theoretical impact of the key parameter, λ. A real-world app misbehavior might occur intermittently (e.g., Figure 2). The impact of λ for such intermittent misbehavior

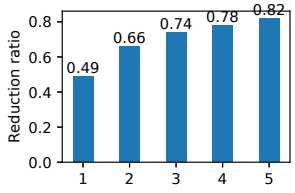

Figure 12. Reduction ratio of power waste with different λ

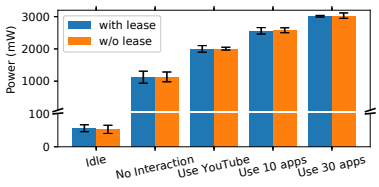

Figure 13. Power consumption overhead of LeaseOS under five settings.

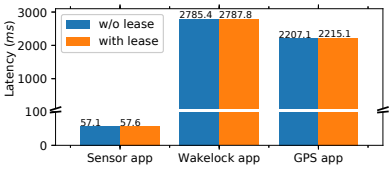

Figure 14. Average end-to-end latency (*ms*) for three representative apps.

cannot be easily captured with a formula. We wrote a test app to simulate the impact. The test app generates 1000 misbehavior slices and 1000 normal slices, each with a random length from 0 to 10min. The combination of the slices is a test case. We generate 1000 test cases and calculate their reduction ratio under different λ from 1 to 5. Figure 12 shows that the larger λ is, the higher reduction ratio. But a larger λ also increases the probability of misjudging a normal behavior as low-utility and the expected penalty (slow-down for legitimate app execution).

7.6 Overhead

We measure the system power consumption overhead of LeaseOS with the Monsoon power monitor. We evaluate five settings: 1) idle with only stock apps and screen off; 2) no user interactions but with screen on and a number of popular apps installed; 3) use YouTube; 4) use 10 apps in turn; 5) use 30 apps in turn. Each experiment is run 8 times. For experiments involving using apps, we try to repeat the user interactions across runs with best efforts. Figure 13 shows the average results with error bars for LeaseOS and the vanilla Android. We can see that LeaseOS introduces negligible overhead (< 1%), with a slightly larger variance. The low overhead is a result of the lightweight lease proxies and the adaptive lease term optimizations.

As an end-to-end test, we measure the energy consumption with one buggy GPS app in the system. In the experiment, we play music for 2 hours, watch YouTube for 1 hour, browse for 30 mins and keep the phone on standby. The result shows that Android w/o lease runs out of battery after around 12 hours, while LeaseOS lasts for 15 hours.

We also measure the impact of lease to end-to-end app latency. We choose three representative apps with interaction flows (e.g., button click to UI updates) that involve resource backed by the lease. Figure 14 plots the latency results, which show that lease introduces very small latency overhead.

8 Discussion

While our evaluation shows lease is an effective mechanism in addressing real-world energy misbehavior, there are several limitations that we plan to address as our future work.

Our utility metrics are currently defined and measured based on the assumption that the energy consumption is proportional to the duration of resource usage. To account for complex hardware behavior such as Dynamic Voltage and Frequency Scaling (DVFS), we need to adjust the metrics with device state factors.

As a system-level solution, LeaseOS does not understand the semantics of apps and uses a set of generic utility metrics instead. The lack of semantic information could lead to potential misclassification for certain apps. Writing custom utility functions in these cases becomes necessary. But we think this effort is still acceptable compared to the efforts needed for energy-efficient programming.

LeaseOS also cannot draw a clear line between normal usage and the excessive usage because they both appear to the lease manager as having high utility. We plan to investigate inferring app and user intentions as part of the utility measurement to tackle the *Excessive-Use* behavior.

The lease policies and parameters are statically set based on the offline analysis (§3.3 and §5.1) There may be new misbehavior patterns that cannot be addressed by the current misbehavior judging policies. We plan to adjust the policies dynamically based on app usage history in the future.

9 Related Work

OS Support for Energy Efficiency. Resource-constrained mobile devices require special OS support. A plenty of systems [33, 42, 48, 49, 61, 62, 64, 68] are designed for efficient mobile resource management. To name a few, Anand et al. [33] propose OS interfaces to expose "ghost hints" from applications to devices; ECOSystem [68] proposes the unified *currentcy* model to provide fair energy allocations; JouleGuard [42] uses control theory to provide energy guarantees for approximate applications. These systems aims to optimize energy under normal conditions while our system's goal is to reduce energy waste due to defects in apps.

Cinder [61] proposes new abstractions—reserves and taps—to explicitly control energy consumption. The reserve in Cinder and our proposed lease abstraction both describe the right to use a resource. But Cinder treats the reserve as an allotment; thus, an application granted with a reserve can run as long as the allotment is not exceeded, even if the energy is wasted. Our lease describes rights to fine-grained kernel resources in the temporal dimension. App granted

with a lease can continue to use the resource as long it makes efficient use of it.

Energy-Aware Adaptation. Building feedback loop into mobile system is a well-studied approach to achieve energy efficiency (e.g., Odyssey [39, 54], Grace OS [67], Proportion allocator [65], SPECTR [59], CALOREE [53]). These solutions typically work by monitoring resources, energy and environment changes to adapt and tune application behavior accordingly. They assume the mobile system and applications are collaborative. We target the scenarios where apps can misbehave, and use utilitarian leases to reward apps that can efficiently use resources.

Runtime Mitigation. Several runtime solutions for reducing energy consumption of background apps exist [11, 45, 52, 63]. Among them, Doze [28] and DefDroid [45] are most closely related to LeaseOS. Doze extends battery life by deferring background CPU and network activity when the device is not used for a long time. DefDroid applies fine-grained throttling on disruptive apps when certain resources are held for too long. Such one-shot deferral or throttling approach based on holding time cannot distinguish misbehavior from legitimate heavy resource usage. Therefore they can easily overreact to normal apps. By using lease with the utility metrics, LeaseOS inherently incurs little negative usability impact. The continuous *examine-renew* also allows LeaseOS to adapt well for intermittent energy misbehavior (e.g., due to environment conditions).

Energy Bug Detection and Diagnosis. App energy misbehavior is a common issue that frustrates many users. Pathak et al. [58] first study the code patterns of no-sleep energy bugs and propose static analysis solution to detect these bugs. A number of subsequent projects have improved the app bug detection techniques [34, 43, 44, 46], fine-grained power profiling [36, 37, 56, 57], app testing [51], and diagnosis of abnormal battery drain [50]. LeaseOS focuses on runtime mitigation of apps with energy bugs, which is complementary to these work.

10 Conclusion

With abundant apps available on the energy-constrained mobile devices, it becomes ever more important to design resource management mechanism that can mitigate app energy misbehavior at runtime. We explore a utilitarian approach in the design space for mobile resource management, and propose the mobile lease abstractions. We design LeaseOS that continuously measures the utility of app resources to make lease decisions. Experiments show that LeaseOS can reduce the wasteful power consumption for 20 real-world buggy apps by 92% on average, significantly more effective than two state-of-the-art runtime solutions. Experiments also show that for apps that use resources heavily for legitimate purpose can properly function under LeaseOS due to its inherent utilitarian nature. LeaseOS incurs <1% power consumption overhead.

The source code of LeaseOS is publicly available at:

https://orderlab.io/LeaseOS

Acknowledgments

We thank the anonymous reviewers for their valuable suggestions that substantially improved this paper.

References

[1] Only lock Wi-Fi if our active network is Wi-Fi upon connection. https://github.com/connectbot/connectbot/commit/b7cc89c811bf07240251b746a59a292dcfea7ec5, 2008.
[2] Release the wakeLock in onPause(), because onpause is guaranteed to be called. https://github.com/jwood/standup-timer/commit/72bf4b96e7ba9457a359c4c8a4b4f653d238fab9, 2009.
[3] Can't find location message was still posted even if location manager was turned off. https://github.com/bostonbusmap/bostonbusmap/commit/9fa09e7b414fe3d88fbac56ad3ec077117a402ca, 2010.
[4] connectbot issue #299. https://code.google.com/archive/p/connectbot/issues/299, 2010.
[5] Facebook confirms battery drain in latest android build. https://www.androidcentral.com/facebook-confirms-battery-drain-latest-android-update, 2010.
[6] Fixed battery drain and delete messages. https://github.com/mvitaly/k-9/commit/4542e64bc612a7e4868ae4f3b9cace7fa81f3c52, 2011.
[7] FlashDevice: Get the wakelock only if it isn't held already. https://github.com/CyanogenMod/android_packages_apps_Torch/commit/2d5c64cee5ecd825a7cb21ec4cbf09ea148cc5e7, 2011.
[8] New feature: Location accuracy. https://github.com/mendhak/gpslogger/issues/4, 2011.
[9] opengpstracker issue #239. https://code.google.com/p/open-gpstracker/issues/detail?id=239, 2011.
[10] Save power when not connected to an access point. https://github.com/servalproject/batphone/issues/50, 2012.
[11] Amplify. http://forum.xda-developers.com/xposed/modules/mod-nlpunbounce-reduce-nlp-wakelocks-t2853874, 2014.
[12] Battery consumption way too high. https://github.com/SecUpwN/Android-IMSI-Catcher-Detector/issues/87, 2014.
[13] Battery draining. https://github.com/kontalk/androidclient/issues/143, 2014.
[14] GPS stays active. https://github.com/opensciencemap/vtm/issues/31, 2014.
[15] High battery drain with no gps lock. https://github.com/MarcDufresne/BetterWeather/issues/6, 2014.
[16] Interval based peroidic scanning. https://github.com/mozilla/MozStumbler/issues/369, 2014.
[17] iOS 7 background multitasking. https://www.captechconsulting.com/blogs/ios-7-tutorial-series-whats-new-in-background-multitasking, 2014.
[18] Battery usage is high. https://github.com/WhisperSystems/TextSecure/issues/2498, 2015.
[19] Facebook ios app battery drain in October 2015. https://www.facebook.com/arig/posts/10105815276466163, 2015.
[20] Polls sensors even when screen is off. https://github.com/gabm/TapAndTurn/issues/28, 2017.
[21] Riot-im, google play version 0.7.07 accelerometer use. https://github.com/vector-im/riot-android/issues/1830, 2017.
[22] Android forums. https://androidforums.com/, 2018.
[23] BetterWeather app. https://play.google.com/store/apps/details?id=net.imatruck.betterweather, 2018.
[24] Github. https://github.com, 2018.

[25] Google code. https://code.google.com, 2018.
[26] K-9 Mail app. https://k9mail.github.io/, 2018.
[27] Kontalk messaging app. https://kontalk.org, 2018.
[28] Optimizing for Doze and app standby in Android. http://developer.android.com/training/monitoring-device-state/doze-standby.html, 2018.
[29] Qualcomm Trepn profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler, 2018.
[30] Volley library for android apps. https://github.com/google/volley, 2018.
[31] XDA developers. http://www.xda-developers.com, 2018.
[32] UI/Application exerciser monkey. https://developer.android.com/studio/test/monkey.html, 2019.
[33] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in the machine: interfaces for better power management. In *Proc. of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, pages 23–35, Boston, Massachusetts, 2004.
[34] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 588–598, Hong Kong, China, 2014.
[35] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pages 151–164, Portland, Oregon, USA, 2015.
[36] X. Chen, J. Meng, Y. C. Hu, M. Gupta, R. Hasholzner, V. N. Ekambaram, A. Singh, and S. Srikanteswara. A Fine-grained Event-based Modem Power Model for Enabling In-depth Modem Energy Drain Analysis. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, pages 107–109, Irvine, CA, USA, 2018.
[37] N. Ding and Y. C. Hu. GfxDoctor: A holistic graphics energy profiler for mobile devices. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 359–373, Belgrade, Serbia, 2017.
[38] U. Farooq and Z. Zhao. RuntimeDroid: Restarting-free runtime change handling for android apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pages 110–122, Munich, Germany, 2018.
[39] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 48–63, Charleston, South Carolina, USA, 1999.
[40] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210. ACM, 1989.
[41] D. Hackborn. Comments on Android OS check for excessive wake lock usage. https://www.mail-archive.com/android-developers@googlegroups.com/msg138995.html, 2011.
[42] H. Hoffmann. JouleGuard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214, Monterey, California, 2015.
[43] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, Edinburgh, United Kingdom, 2014.
[44] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 18:1–18:15, Amsterdam, The Netherlands, 2014.
[45] P. Huang, T. Xu, X. Jin, and Y. Zhou. DefDroid: Towards a more defensive mobile os against disruptive app behavior. In *Proceedings of the The 14th ACM International Conference on Mobile Systems, Applications, and Services*, Singapore, Singapore, June 2016.
[46] X. Jin, P. Huang, T. Xu, and Y. Zhou. Nchecker: Saving mobile app developers from network disruptions. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 22:1–22:16, London, United Kingdom, 2016.
[47] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Proceedings of the 7th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '13, pages 15–24, Oct 2013.
[48] M. Lentz, J. Litton, and B. Bhattacharjee. Drowsy power management. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 230–244, Monterey, California, 2015.
[49] D. Liaqat, S. Jingoi, E. de Lara, A. Goel, W. To, K. Lee, I. De Moraes Garcia, and M. Saldana. Sidewinder: An energy efficient and developer friendly heterogeneous architecture for continuous mobile sensing. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 205–215, Atlanta, Georgia, USA, 2016.
[50] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 57–70, Lombard, IL, 2013.
[51] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, Saint Petersburg, Russia, 2013.
[52] M. Martins, J. Cappos, and R. Fonseca. Selectively taming background Android apps to improve battery lifetime. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 563–575, Santa Clara, CA, 2015.
[53] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann. CALOREE: Learning control for predictable latency and low energy. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 184–198, Williamsburg, VA, USA, 2018.
[54] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 276–287, Saint Malo, France, 1997.
[55] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, Cambridge, Massachusetts, 2011.
[56] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, Bern, Switzerland, 2012.
[57] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 153–168, Salzburg, Austria, 2011.
[58] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, Low Wood Bay, Lake District, UK, 2012.
[59] A. M. Rahmani, B. Donyanavard, T. Mück, K. Moazzemi, A. Jantsch, O. Mutlu, and N. Dutt. Spectr: Formal supervisory control and coordination for many-core systems resource management. In *Proceedings of the Twenty-Third International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS '18, pages 169–183, Williamsburg, VA, USA, 2018.

[60] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 190–203, Bretton Woods, New Hampshire, USA, 2014.

[61] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the Cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 139–152, Salzburg, Austria, 2011.

[62] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power Containers: An OS facility for fine-grained power and energy management on multicore servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 65–76, Houston, Texas, USA, 2013.

[63] I. Singh, S. V. Krishnamurthy, H. V. Madhyastha, and I. Neamtiu. ZapDroid: Managing infrequently used applications on smartphones. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 1185–1196, Osaka, Japan, 2015.

[64] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: A platform for OS-level power management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 289–302, Nuremberg, Germany, 2009.

[65] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 145–158, New Orleans, Louisiana, USA, 1999.

[66] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal. Towards verifying Android apps for the absence of no-sleep energy bugs. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, pages 3–3, Hollywood, CA, 2012.

[67] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 149–163, Bolton Landing, NY, USA, 2003.

[68] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 123–132, San Jose, California, 2002.