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

ASPLOS Submission #404– Confidential Draft – Do Not Distribute!

Abstract

Mobile apps have become indispensable in modern life. However, many mobile apps are not designed to be energy-aware that it may hold an energy-consuming resource for long even after the resource is no longer useful to the app. Blindly restricting apps' use of resources, while helps reduce energy consumption, prohibits apps from taking advantages of the resources to do useful work. The fundamental missing piece is a feedback loop in the OS resource management to regularly assess if a resource is still needed 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 LeaseOS, a lease-based resource management mechanism that takes a utilitarian approach to continuously analyze the utility of a resource to an app at each lease term, and then make 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 did not cause any usability disruption to the apps. LeaseOS itself incurs negligible energy overhead.

1. Introduction

Throughout the past decade, mobile devices have transformed from tightly-closed systems with only a handful of proprietary software to open platforms that offer massive programmability for third-party developers to write apps. For example, the Android 7 SDK provides 30,662 APIs. However, programming mobile devices is not easy. The mobile programming model is inherently event driven, which complicates the app code reasoning, execution flows and testing. App developers have to take into account the constantly changing environment and make careful optimizations for various constrained resources. For many app developers who are unseasoned in system programming, these requirements are especially challenging.

As a result, a lot of mobile apps are buggy. A common and severe type of bug is energy bug that leads to abnormal battery drain [13, 25, 30]. 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, many apps may make an `acquire` request and forget to release the resource in some code paths or release it very late, which causes severe battery drain. Similarly, iOS manages the audio resource via audio sessions to apps. The Facebook iOS app had a buggy release [3] that would leak the audio sessions in some scenarios, leaving the app doing nothing but staying awake in the background draining battery fast. That release also had a bug in the network handling code that would incur long CPU spins without making any progress.

Solutions have been proposed to remove energy bugs before release with better app testing [9, 26, 31], bug detection [21, 23, 28, 34], and libraries [10], but complex bugs can still escape these tools. Thus it is important to design techniques to mitigate energy defects at runtime. After all, energy bugs can cause damage at user side in part because existing resource management mechanisms are insufficiently protective.

State-of-the-art runtime techniques [4, 18, 22, 27] monitor app resource usage, and kill or throttle apps if the hold time of some resource exceeds a threshold. However, holding a resource for long does *not* necessarily mean misbehavior. There are legitimate scenarios where apps do need to make heavy use of resources, e.g., for navigation or gaming. Blind throttling therefore could significantly disrupt app functionalities. The fundamental missing piece in mobile resource management is a feedback loop mechanism to assess whether a resource is still truly useful to the app *after* the resource is granted.

This paper proposes that lease [17], 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, read accesses at the client do not need approval from the server. After the term, if the lease is not renewed, e.g., due to client crash or network partition, the server can safely proceed instead of waiting indefinitely.

Although energy misbehavior in mobile devices is a different problem domain, the essential abstraction of lease can be extended to mobile settings as a contract between the mobile OS and an app about a resource (e.g., wakelock, GPS, sensor) with a condition on time. Such abstraction brings two benefits. First, leases can tolerate sloppy resource use 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, which 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 to mitigate app energy misbehavior. In LeaseOS, a resource granted to an app can be backed by a lease with a term from zero to infinity. If the app still holds the resource at the end of the term, the lease will be either extended or deferred (temporarily expired). An attempt to use resource with an expired lease later requires approval by the lease manager.

A main challenge in the design of LeaseOS is to make informed lease decisions, e.g., whether to renew a lease and the length of a term. The ideal lease decisions should effectively mitigate energy misbehavior, but should *not* deprive apps of legitimate heavy use of resources. A straightforward decision policy is to use the absolute resource hold time [22, 27]. But through studying real-world apps (Section 2), we find this metric is a misleading classifier for energy misbehavior.

LeaseOS instead takes a utilitarian approach to make 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 break down energy misbehavior into four classes: namely, (a) *Frequent-Ask*, but the resource is rarely granted, (b) *Long-Holding*, but the granted resource is rarely used, (c) *Low-Utility*, the granted resource is used to do a lot of work, but most of the work is useless, e.g., generating repeated `IOExceptions`, (d) *Excessive-Use*, resource is used to do useful work but it incurs high overhead. We argue that (d) is where the line between legitimate and selfish behavior starts to blur, but the first three types can be effectively identified and mitigated, which are the main targets of LeaseOS.

There are two possible flavors of lease designs. One is app-oblivious lease, in which all lease operations including creation and renewal happen behind the scene. The other is lease with APIs to involve apps in the lease management, e.g., direct renew requests, expiration callbacks. The first choice requires no changes to the app source code. But the transparency may make suboptimal lease decisions in complex scenarios. The second choice leverages app semantics. But developers might be overwhelmed with using lease APIs. The excessive developer efforts could even re-introduce energy misbehavior (e.g., simply call the renew API unconditionally in the expiration callback). LeaseOS mainly uses app-oblivious lease management and calculates a set of generic utilities using metrics guided by our study on real-world apps (Section 2). To leverage app semantic information without overburdening developers, LeaseOS provides one simple API for apps to 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.

Another challenge that LeaseOS must address is performance. Because mobile apps are latency critical, the lease mechanism should not introduce significant delays to apps or battery drain itself. LeaseOS introduces efficient data structures and a small set of light-weight *lease proxies* that bypass unnecessary communications with the lease manager. Additionally, we design adaptive lease terms to optimize for the common cases, where apps are behaving normally so the lease terms will be long enough to avoid the overheads in lease management and tracking.

We implemented LeaseOS on the latest 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 [7] and DefDroid [22], 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.

2. Understanding Energy Misbehavior

To improve existing mobile resource management, it is useful to first understand the runtime characteristics of app energy misbehavior. We reproduced and analyzed five real-world buggy apps on multiple smartphones and environments. In this Section, we highlight three cases and our main insights.

2.1. Real-world Cases and Experiment Setup

Case I. K-9 mail [5] is a widely-used Android email app. The app has a defect that when the network is disconnected or when the mail server fails, the app will encounter an exception and handles the exception by retrying indefinitely. For each time of retrying, the app acquires a wakelock, which causes severe battery drain. The issue was fixed by adding an exponential back-off and prompt wakelock release.

Case II. Kontalk [6] is a popular messaging app. When a user logs in, Kontalk authenticates with a server and establishes a connection. In one release, Kontalk 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 and causes excessive battery drain. The defect is fixed by releasing the wakelock as soon as the app is authenticated.

Case III. BetterWeather [2] 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 signal. The bug was fixed by giving up the search after 30 seconds.

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` +

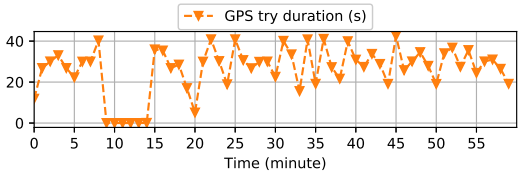

Figure 1: BetterWeather's GPS try duration every 60s.

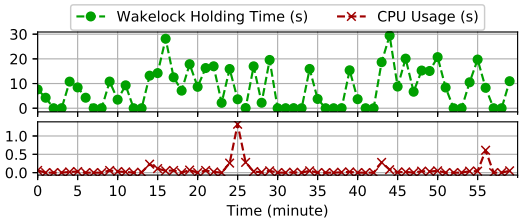

Figure 2: Wakelock holding time and CPU usage of buggy K-9 mail in a connected environment with a bad mail server.

`userTime`). The experiments are run on four different Android phones: 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 Samsung and Moto phones being heavily used and the Nexus phones 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?

The commonly used energy misbehavior characteristic is that certain energy-consuming resource such as active CPU, GPS, sensors is excessively used. Therefore, the corresponding mitigation is to monitor the resource usage and throttle apps if the usage is excessive. But such characterization is too ambiguous to distinguish active resource use. From studying the code patterns of energy bug cases, we find that an intrinsic characteristic of energy misbehavior lies in an abstract model in the existing mobile resource management: *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. Experiment 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 signal. Figure 1 shows the GPS request duration over time. Each point is measured ever 60s. We can see that for each measurement interval, the app spends around 60% of the time asking for the GPS lock. But since the app never get 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 do the main work, 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.

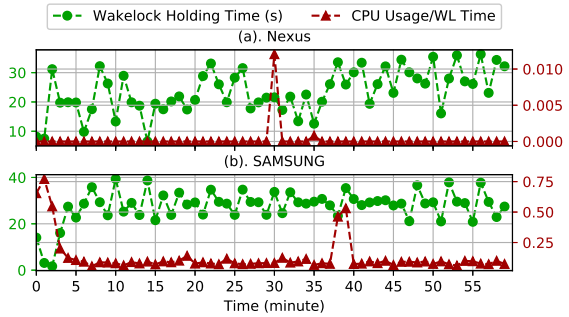

Figure 3: The wakelock holding time and ratio of CPU usage over wakelock time for the buggy Kontalk app on two phones.

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 plots the result, which shows in majority of the one-minute measurement intervals, the wakelock holding time is long. However, 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 $2\times$ because of 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 often flag a normal app as misbehaving. A 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 process systime or usertime should be consistent with 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 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

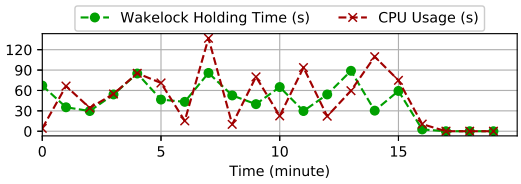

Figure 4: The wakelock holding time and CPU usage of buggy K-9 mail in a network-disconnected environment.

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 for this condition. We can see that, compared to the results in 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.

2.4. Energy Misbehavior Classification

Based on the experiments analysis, Table 1 summarizes four types of energy misbehavior in the *ask-use-release* model. The first three are due to clear defects in the app code: *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 categories apply to seven types of 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

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

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.

do anything including not using it. But for GPS or sensors, 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% because the system handles it. We define the LHB for GPS/sensor as the utilization of the GPS location *data* rather than the physical resource.

The previous experimental observations give us insights on how to classify app resource usage behavior for making lease decisions. When energy defects manifest themselves at runtime, the resource holding time typically increases. But the absolute time alone is not a reliable indicator that can separate out normal behavior. The unique characteristic we observe across different apps and ecosystems centers around how well a resource is utilized. In particular, when energy defects are triggered, the resource acquire 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 calculating and checking the resource utility metrics at the end of a lease term is sufficient to catch energy misbehavior early on. Compared to the alternative of sub-dividing the lease term into very small epochs and checking after each epoch, checking at the end of an entire lease term reduces overhead.

3. Lease Abstraction for Mobile System

This Section describes the core abstraction in LeaseOS, leases, and the motivation for changing the *ask-use-release* resource management mechanism in mobile devices to be lease-based.

3.1. Abstraction

Existing OS provides APIs for an app to use various resources. After an initial sanity check, the right to request and hold the resource persists indefinitely unless the app explicitly renounces it. This model assumes that apps are capable of efficiently managing resources throughout the resource lifetime, which is problematic. Take the buggy K-9 mail app as an example. The granted wakelock is used to retry non-stop in a network disconnected environment that leads to high energy waste.

LeaseOS explicitly manages the resource rights by introducing leases. A lease essentially grants an app the rights to

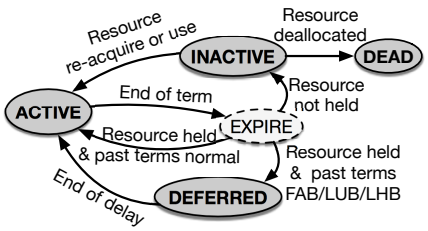

Figure 5: Lease state transition.

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*. Under 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 to judge if the resource is efficiently used. 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, 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 makes a resource release call 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 with expired lease, the access requires a check with the OS who will make a renewal decision. When the resource covered by the lease 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

To allow LeaseOS 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 signal, 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 utilization ratio is 100%. We find that 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 for GPS and sensor. *Low-Utility* behavior refers to the utilization rate of a granted resource is high but most of 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. Take a screen control app an example, the app provides an icon for user 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.

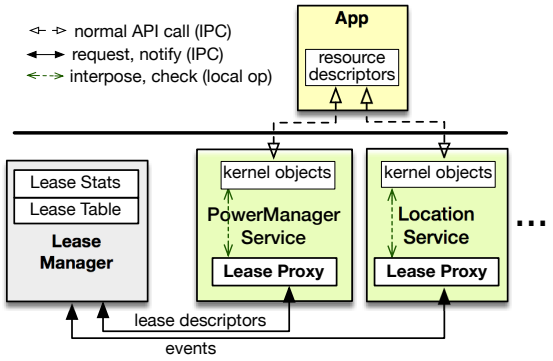

Figure 6: Architecture of LeaseOS.

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 most user complains on app energy misbehavior fall into FAB, LUB or LHB, which are due to clear programming mistakes. EUB, on the other hand, is caused by heavy use of resources, which is controversial to judge as a misbehavior.

4.1. Overview

Figure 6 shows the architecture of LeaseOS. A centralized 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 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 interact with the subsystems via IPCs to make resource requests (e.g., creation, update) via 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 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 lease table, which 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 the 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 2 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 lease holder (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 ob-

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

Table 2: LeaseOS interfaces for lease proxies. `setUtility` is exposed to apps.

jects) 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 7 uses a real-world app (K-9 mail) as an example to show the lease mechanism from 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

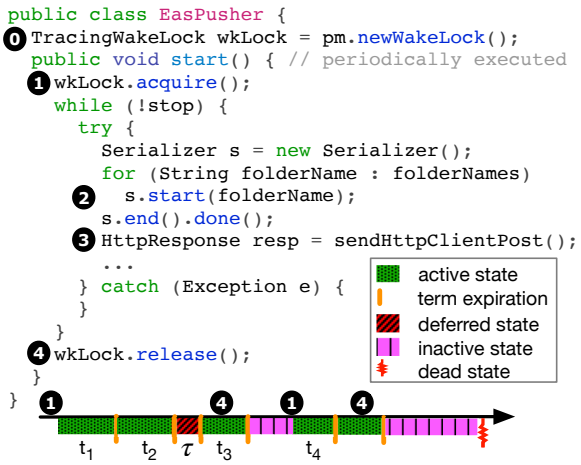

Figure 7: Lease mechanism from an app's perspective.

containing ❹ finishes, the lease manager will find 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 will immediately request the manager to remove the lease. During the entire process, even if the absolute holding time between ❶ and ❹ is long, the app will behave the same as without lease mechanism.

The app may exhibit an 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.

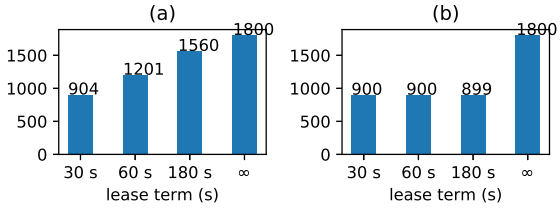

Figure 8: Resource holding times (s) of a test buggy app with Long-Holding misbehavior under different lease terms.

But the resource descriptor in 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 7 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. 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 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. For our target scenario, the false sharing trade-off does not exist because most energy-consuming mobile resources are shared in a subscription style that do not require coordination. But lease term, t, as well as the deferral interval, τ, that we introduced in LeaseOS, influence the effectiveness on mitigating energy misbehavior and impact to legitimate resource usage. A short lease term allows the lease manager to quickly detect an energy misbehavior. But it can incur high lease accounting overhead. A short deferral interval can reduce the negative impact (slowdown of legitimate high-utility executions) on misjudgment. But it has limited effect on reducing wasteful energy consumption.

We analyze the relationship between the effectiveness of mitigating misbehavior and lease term. Suppose that an app starts to exhibit the Long-Holding misbehavior at the beginning of ith lease term as keeping holding the resource while doing nothing, 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 8 (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 again with the same three lease terms but keep λ to be 1. Figure 8 (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 relative 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 manage will increase the lease term to 1

minute if the past 12 terms (1 minute) are normal, and further increase it to 5 minute 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 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. Because apps are forked from Zygote, the libcore is shared among apps. Our initial exception tracker is a singleton in libcore. Unfortunately, the system services' exception trackers are separate from the apps' so the framework cannot get the updated information. 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 [8] 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, we use a Nexus 5X phone as a substitute to setup the measurement (Figure 11). To make sure the baseline Android system has the same app ecosystem, instead of re-flashing a vanilla factory image each time, we provide a flag in LeaseOS to completely turn off the lease manager service and proxies. The *initial* lease term used is 5 seconds with default deferral interval of 25 seconds.

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

Table 3: Average latency of major lease operations in *ms*.

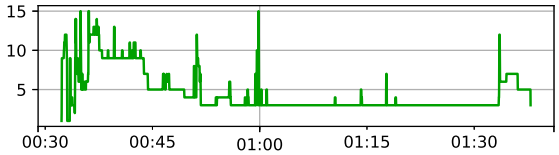

Figure 9: Number of active leases in one hour period

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 for 20 times and then collect the latencies for different operations. Table 3 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 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 9 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 work. Starting from 6.0, Android introduces the Doze mode [7] which defers app background CPU and network activity when the device is unused for a long time. Amplify [1] and DefDroid [22] throttle excessive requests. We compare the mitigation effectiveness of LeaseOS with Doze and the simple throttling approach (with the throttling settings the same as in

App	Category	Res.	Behavior	Power (mW)				Reduction Percent (%)		
				w/o lease	w/ lease	Doze*	DefDroid	LeaseOS	Doze	DefDroid
Facebook	social	CPU	LHB	100.62	1.93	18.92	12.68	98.08	81.19	87.40
Torch	tool	CPU	LHB	81.54	1.30	19.26	14.39	98.41	76.38	82.35
Kontalk	messaging	CPU	LHB	29.41	0.39	16.84	15.99	98.67	42.74	45.63
K-9	mail	CPU	LUB	890.35	81.62	195.2	136.14	90.83	78.08	84.71
ServalMesh	tool	CPU	LUB	134.27	1.37	30.54	14.88	98.98	77.25	88.92
TextSecure	messaging	CPU	LUB	81.62	1.198	18.78	16.78	98.53	76.99	79.44
ConnectBot	tool	screen	LHB	576.52	23.23	573.23	115.56	95.97	0.57	79.96
Standup Timer	productivity	screen	LHB	569.10	13.26	544.46	61.82	97.67	4.33	89.14
ConnectBot	tool	Wi-Fi	LHB	17.08	0.78	3.21	2.57	95.43	81.21	84.95
BetterWeather	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	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	travel	GPS	LHB	118.25	4.33	21.98	70.7	96.34	81.41	40.21
BostonBusMap	travel	GPS	LHB	115.5	3.97	19.5	71.09	96.56	83.12	38.45
AIMSCID	service	GPS	LUB	119.43	4.50	23.91	73.31	96.23	79.98	38.62
OpenScienceMap	navigation	GPS	LUB	123.97	3.40	19.91	91.25	97.26	83.94	26.39
OpenGPSTracker	travel	GPS	LUB	360.25	1.32	19.91	237.41	99.63	94.47	34.10
TapAndTurn	tool	sensor	LUB	11.72	1.87	3.95	4.41	84.04	66.30	62.37
Riot	messaging	sensor	LUB	19.17	1.43	6.64	3.93	92.54	65.36	79.50
							Average:	92.62	69.64	62.04

Table 4: 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.

DefDroid [22]). Each experiment is run for 30 minutes, during which the power consumption is sampled every 100*ms*.

Table 4 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 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 4 results show 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 analyze 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 is by design optimizing for apps' utility. The three types of misbehavior

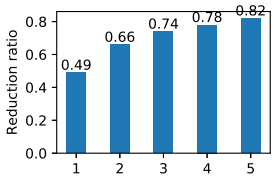

Figure 10: Reduction ratio of power waste with different λ.

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 monitor 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 the time-based throttling scheme, all of the three apps experienced some disruption, e.g., fitness tracking,

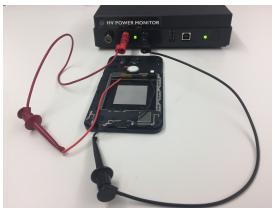

Figure 11: Nexus 5X with Monsoon power monitor.

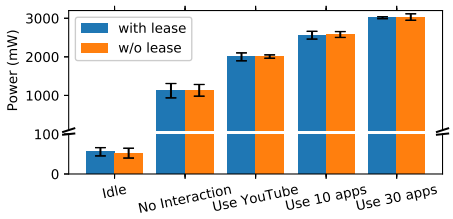

Figure 12: Power consumption overhead of LeaseOS under five settings.

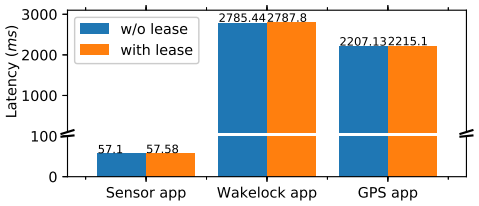

Figure 13: Average end-to-end latency (*ms*) for three representative apps.

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 a 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 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 10 plots the results, which show 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 (degree of delay or 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 12 shows the average results with error bars for LeaseOS and the vanilla Android. We can see that LeaseOS introduces negligible overhead ($< 1\%$), with a slight larger variance. The low overhead is a result of the lightweight lease proxies and the adaptive lease term optimizations for the common case.

We also measure the impact of lease to end-to-end app latency. We choose three representative apps with interaction flows (e.g., from a button click to UI updates) that involve resource backed by lease. Figure 13 plots the latency results, which show that lease introduces very small latency overhead.

8. Related Work

Resource-constrained mobile devices require special OS support. A plenty of systems [11, 19, 24, 32, 35] are designed for efficient mobile resource management. Anand et al. [11] propose OS interfaces to expose "ghost hints" from applications to devices; ECOSystem [35] proposes the unified *currentcy* model to provide fair energy allocations; JouleGuard [19] uses control theory to provide energy guarantees for approximate applications. Our work proposes using the lease abstraction to manage resources on mobile devices. Our goal is to mitigate energy waste due to defects in apps, rather than to optimize energy under normal conditions.

App energy misbehavior is a common issue that frustrates many users. There is a body of work that characterizes and detects app energy bugs [12, 16, 20, 21, 23, 30], provides fine-grained power profiling [14, 15, 28, 29], improves app testing coverage [26] and automates the diagnosis of abnormal battery drain [25]. LeaseOS focuses on runtime mitigation of apps with energy bugs, which is complimentary to these solutions.

Several runtime solutions for reducing energy consumption of background apps exist [1, 22, 27, 33]. Among them, Doze [7] and DefDroid [22] 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).

9. Conclusion

LeaseOS is a utilitarian resource management mechanism for energy-constrained mobile devices to mitigate app energy misbehavior. At its core, LeaseOS uses lease abstractions. The lease decisions are made based on continuously measuring the utility of resources. 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 but for legitimate purpose can properly function under LeaseOS due to its inherent utilitarian nature. But the existing blind deferral or throttling solutions can impair the functionality of these apps. LeaseOS incurs $<1\%$ power consumption overhead.

References

[1] Amplify. http://forum.xda-developers.com/xposed/modules/mod-nlpunbounce-reduce-nlp-wakelocks-t2853874.

[2] BetterWeather app. https://play.google.com/store/apps/details?id=net.imatruck.betterweather.

[3] Facebook ios app battery drain in October 2015. https://www.facebook.com/arig/posts/10105815276466163.

[4] iOS 7 background multitasking. https://www.captechconsulting.com/blogs/ios-7-tutorial-series-whats-new-in-background-multitasking.

[5] K-9 Mail app. https://k9mail.github.io/.

[6] Kontalk messaging app. https://kontalk.org.

[7] Optimizing for Doze and app standby in Android. http://developer.android.com/training/monitoring-device-state/doze-standby.html.

[8] Qualcomm Trepn profiler. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.

[9] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html.

[10] Volley library for android apps. https://github.com/google/volley.

[11] 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.

[12] 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 2014, pages 588–598, Hong Kong, China, 2014.

[13] 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.

[14] 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.

[15] 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.

[16] N. Ding, D. Wagner, X. Chen, A. Pathak, Y. C. Hu, and A. Rice. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 29–40, Pittsburgh, PA, USA, 2013.

[17] 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.

[18] D. Hackborn. Comments on Android OS check for excessive wake lock usage. https://groups.google.com/forum/#!topic/android-developers/Tg7_sUL8Ec4.

[19] 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.

[20] 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.

[21] 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.

[22] 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*, June 2016.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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 *Proc. of the sixth conference on Computer systems*, EuroSys '11, pages 139–152. ACM, 2011.

[33] 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.

[34] 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.

[35] 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.