DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior

Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou
University of California, San Diego

ABSTRACT

The mobile app market is enjoying an explosive growth. Many people, including teenagers, set about developing mobile apps. Unfortunately, due to developers' inexperience and the unique mobile programming paradigms, a growing number of immature apps are released to users. Despite having useful functionalities, these apps exhibit *disruptive* behaviors that are inconsiderate to the mobile system as a whole, *e.g.*, retrying network connections too aggressively, waking up the device too frequently, or holding resources for unnecessarily long. These behaviors adversely affect other apps running on the same device and frustrate users with battery drain, excessive cellular data consumption, storage overuse, *etc.*

In this paper, we investigate *Disruptive App Behavior (DAB)* with a study on 287 real-world DAB issues. Guided by the study, we present DefDroid, a mobile OS designed to protect users from the negative impact of DAB at runtime. DefDroid monitors important app activities and tries to adjust app behaviors using fine-grained actions (*e.g.*, enforce back-off to continuous retries, decrease aggressive timer frequency) without breaking app main functionality. Our experiments show that DefDroid effectively curbs 125 real-world DAB cases with small overhead and marginal impact to the usability of both the misbehaving apps and normal apps. During a small-scale user trial, DefDroid also found 6 new DAB issues. We further deployed DefDroid to 185 real users through the PhoneLab testbed for 43 days and found DAB issues from at least 57 apps.

CCS Concepts

•Computer systems organization → Reliability; •Software and its engineering → Operating systems; •Human-centered computing → *Mobile computing;*

Keywords

Mobile apps; Operating system; Defensive actions

MobiSys'16, June 25-30, 2016, Singapore, Singapore

DOI: http://dx.doi.org/10.1145/2906388.2906419

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider
  android:minHeight="146dp"
  android:minWidth="146dp"
  android:updatePeriodMillis="2500"/>
```

Figure 1: Anki-Droid sets an aggressive widget update interval (2.5s) that causes severe battery drain [17].

1. INTRODUCTION

The booming of mobile users and device shipments [46] has driven a surge of mobile app developers. According to a recent report [37], the mobile developer population has doubled since 2010. In tandem with these growths, abundant mobile apps are released, *e.g.*, up to 60,000 apps are added to App Store monthly [27].

Unfortunately, due to inexperienced app developers and the lack of development resources, many mobile apps are weaker in terms of quality compared to traditional desktop and server applications, especially in dealing with system-level issues such as resource management, exception handling, and network programming. Consequently, many mobile users encounter immature app behaviors (dubbed "naugthy apps") once in a while.

1.1 Disruptive App Behavior

In this work, we attempt to understand and address an emerging class of issues in mobile apps – **Disruptive App Behavior** (*DAB*). DAB refers to apps acting immaturely to its hosting ecosystem and adversely affecting other apps running on the same device.[1] For example, an app may keep the device awake for too long, forget to unregister GPS after use, retry network too aggressively, download large cellular data without user consent, or flood the device screen with noisy notifications. These disruptive behaviors waste or overuse resources without considering other apps, and cause significant battery drain, excessive consumption of bandwidth or storage, *etc.*

DAB issues are mainly caused by programming mistakes and/or aggressive configurations. For example, mobile platform typically provides APIs for apps to schedule periodical task. Some app developers can configure the update frequency to be very aggressive like in Figure 1. As another example, Android provides wakelock APIs [5] to allow apps to prevent hardware components like CPU from entering low-power state. Inexperienced developers often leak an acquired wakelock [52] or hold a wakelock for unnecessarily long [7]. Another common cause for DAB is that developers fail to thoroughly consider and handle changing environment.

[1]We exclude malicious apps that cause security or privacy issues from this definition and our scope of interest

App	Version	Disruptive behavior
F-Droid	0.72	Create "infinite" copies of XML files due to a bug, eating up storage
ownCloud	1.5.4	Sync with server too often and request too much info, consuming large cellular data
Chrome	28.0	Overuse Gyroscope sensor in background, draining battery
AntennaPod	1.2	Display notification for every background podcast download, flooding notification area
Facebook	40.0	CPU spin in network handling, leaking audio session, draining battery
WHERE	3.2.1	Continuously use GPS on standby, draining battery
DAVdroid	0.6	Sync lots of high resolution contact pictures, consuming large cellular data

Table 1: Real-world examples of disruptive app behavior.

```
private requestLocation(LocationManager lm) {
  lm.requestUpdate(provider, mlistener);
+ gpsFixHandler.postDelayed(new Runnable(){
+   public void run(){
+     disableLocationListener();
+   }}, 30 * 1000); // give up after 30s
}
private LocationListener mlistener =
  new LocationListener() {
    public void onLocationChanged() {
+     gpsFixHandler.removeCallbacks(null);
  }
}
```

Figure 2: BetterWeather keeps searching GPS signals and drains battery quickly when users are in a poor-signal environment [6]. Developers fix it by giving up search after 30s.

Figure 2 shows a DAB in a popular weather app that will keep searching for GPS non-stop when its users are in an environment with poor GPS signal. Table 1 further lists several real-world DAB reports.

From app development perspective, the rise of pervasive DAB issues is mainly due to the following reasons:

(1) *Mobile apps are in general "young"*. We examined the top 1,000 apps in two platforms and found that the average age of these apps is 2.5 years for App Store, and 1.5 years for Google Play (Table 2), even though both app stores started in 2008. Therefore, many popular apps have not aged enough to be stable and well-engineered. In comparison, popular desktop/server applications are much more mature: *e.g.*, Firefox is 12 years old and MySQL is 19 years old.

(2) *Mobile app developers also tend to be less experienced and take less rigorous software development training* [64, 63, 57]. Inexperienced app developers often make mistakes in using APIs properly (*e.g.*, calling `release` inside `onStop` instead of `onPause`, forgetting to check network type before transferring data) or inproperly handle various external conditions (*e.g.*, Figure 2). They also tend to uncousiouly assume that their apps will be frequently used by users for a long session, resulting in writing aggressive code or configurations without considering the overall user experiences.

(3) *Many mobile apps are built by a small development team, sometimes individuals*. We find that in App Store, at least 12% of the developers for the top 1,000 apps are individuals (Table 2). Small teams often do not have sufficient resources to conduct comprehensive testing to catch DAB issues before releasing to users, which is especially pressing given that mobile testing faces unique challenges [56].

Platform	Avg. app age	Indie developers
iOS	2.5 years	$> 12\%$
Android	1.5 years	$> 5\%$

Table 2: The average age (time since first release) of the top 1,000 apps in two platforms, and the percent. of independent developers for these apps.

1.2 Diagnosis and Bug Detection Tools

Users who have poor experiences due to DAB often have no clues of why it is happening and how to fix it. Some tech-proficient users leverage advanced tools to diagnose the issues. For example, Android and iOS provide built-in tools to display the battery and data usage by apps. Third-party tools like PowerTutor [68], Wakelock Detector [22], eDoctor [44], and Carat [48] offer more details for troubleshooting.

While these tools can help users find some issues, they suffer from several drawbacks. First, many ordinary users lack the expertise needed to use these tools effectively. Second, these tools are at user level and thus have limited knowledge about app activities. Even when a symptom is successfully diagnosed, the tools need to constantly involve users to take privileged actions for resolution. Most importantly, when these tools are invoked, app misbehaviors usually already cause significant negative impact, *e.g.*, overuse of cellular data. It is desirable to prevent such issues from happening in the first place, or to at least limit their impact to the overall system health.

A number of research work has been proposed to eliminate certain type of app misbehavior using scalable app testing [56, 42], energy bug detection [41, 67, 52, 65], accurate energy profiling [61, 32, 68, 51, 50], and advanced mobile prefetching [39, 55]. Unfortunately, the defects can still escape these tools and be released. Additionally, DAB issues due to aggressive configurations (*cf.*, Figure 1) or complex environment (*cf.*, Figure 2) are not addressed by these tools.

1.3 System-level Support to Prevent DAB

Given that many DAB issues still exist in the field despite fruitful existing tools, it is important to think from mobile system perspective to prevent DAB. One way towards this direction is to redesining existing framework APIs to be less prone to mistakes, which requires developers to rewrite existing apps. This paper focuses on a complementary system-level solution to protect users from DAB without the burden of rewriting apps – runtime mitigation. We argue that mobile system designers should anticipate the common occurrence of DAB during users' daily use of mobile devices, and enhance the OS to deal with DAB at runtime. This not only provides a last defense to DAB for end users but also can

reduce some burden for app developers because the mitigation resembles a hotfix.

Interestingly, we have observed several recent changes in mainstream mobile OSes to address certain DAB issues. For example, after Android OS 2.3, the OS may kill apps for excessive wakelock usage [38]; iOS used to allow apps to run background tasks according to apps' own schedules, but iOS 7 and later versions restrict the actual scheduling to be mainly determined by the OS [23].

Nevertheless, these changes are insufficient and sacrifice usability (*e.g.*, directly killing apps). The OS lacks a comprehensive solution for DAB. As an evidence, users running the latest mobile OS still encounter a significant number of DAB cases, as we will show in §2.

The challenges for designing a comprehensive OS-level solution to tackle DAB are in three folds. First, DAB issues have diverse manifestations, which requires understanding the common patterns to address the issues. Second, the solution needs to target a suspicious app misbehavior without breaking app main functionality. Third, the solution should not get in the way of normal user interactions with apps, and should not incur significant overhead.

1.4 Our Contribution

In this work, we first conduct a study of 287 real-world DAB issues to understand the characteristics of the problem; we then present DefDroid, an Android-based mobile OS designed to restrict the impact of DAB at runtime.

DefDroid modifies Android to monitor important app activities for potential DAB patterns, and, if necessary, dynamically adjust app behavior using *fine-grained defensive actions*. The defensive actions may adjust frequencies, trade off accuracies, change priorities, delay executions, release resources, or warn users. For example, when an app wakes up the device too frequently, DefDroid may adjust the app's alarm intervals; when an app retries network connection too aggressively, DefDroid may enforce a back-off; when an app requests fine locations too frequently in background, DefDroid may start returning coarse locations to that app.

We implement DefDroid based on Android 4.4.3, and evaluate it using 128 real-world DAB cases. The defensive mechanisms in DefDroid successfully limit the negative impact of 125 cases. Compared to unmodified Android, DefDroid reduces the resource consumption impact of the 125 cases in terms of power, mobile data, and storage by up to 481mW, 183MB, and 145MB, respectively, over 30-minute experiment periods.

Our evaluation also shows that the defensive mechanisms in DefDroid cause marginal negative impact to the overall app usability. To be specific, in 40 cases, DefDroid does not induce any side effects to the affected apps. In 80 cases, the defensive actions defer request execution or fail aggressive requests but without causing noticeable impact from users' perspective. For the remaining 5 cases, DefDroid's defensive actions cause checked exceptions that the compilers already enforce developers to handle. There is no crash or abnormal behavior of the test apps caused by defensive actions in our experiments.

During user trials, DefDroid also found and addressed 6 new DAB cases in the field. We have released DefDroid to the PhoneLab testbed [47] and deployed to 185 *real users* for 43 days. Based on logs collected from the deployment, we found DAB issues from 81 apps, 57 of which can be reproduced. We did not receive user complaint about DefDroid breaking app functionality or causing unexpected behavior for any of these apps that DefDroid applied actions to.

Category	# of apps	Downloads	Ratings
Productivity	18	1K–5M	3.2–4.5
Tools	38	10K–50M	3.4–4.8
Social	10	1K–5B	3.5–4.7
Communication	26	5K–5B	3.7–4.6
News	10	1K–5M	3.8–4.6
Media	13	1K–500M	3.5–4.7
Navigation	20	1K–500K	3.6–4.9
Other	46	1K–50M	2.7–4.7

Table 3: Apps in the studied DAB cases.

2. UNDERSTANDING DISRUPTIVE APP BEHAVIORS

To understand the characteristics of DAB, we conduct a study on 287 real-world DAB issues. The purpose of the study is to provide us with insights on solutions to address common DAB. While there are prior studies on certain aspect of DAB (*e.g.*, no-sleep energy bug [52], abnormal battery drain [44]), there is a lack of study on the generic DAB problems as a whole.

2.1 Methodology

We collect the study cases from two sources. First, we manually inspect popular user forums [4, 3, 25] to find posts where users complained about app misbehavior; Second, we examine the issue trackers of open-source mobile apps that have user-reported issues. We obtain an initial set of around 1,000 potential issues from the first source, and 9,200 cases from the second source. We go through each case and filter duplicates. We also ignore misbehavior that only hurts the apps themselves (*i.e.*, crashes, feature bugs).

After filtering, there are **287** cases of DAB from 181 unique apps. Table 3 shows the category and popularity of the apps in the dataset. The majority (265) of studied cases are from open-source apps. This is due to the limitation in our data sources instead of intentional preference. For cases from open-source apps, we know the exact defect causing the DAB. For the remaining 22 cases from closed-source apps, we understand their symptoms and high-level causes.

We also try to reproduce the collected cases on our smartphones running the latest Android OS to further confirm these issues. In total, we successfully reproduced 96 cases. The cases that we failed to reproduce are mainly because of compilation issues, incompatibility, or complex external requirements (*e.g.*, server setup).

2.2 Common Disruptive App Behavior

A first step to understand DAB is to study, both qualitatively and quantitatively, what user-frustrating app misbehaviors affect the ecosystem. Our collected cases are all real user reports/complaints and thus provide a data point to understand this question.

Based on the collected cases, we summarize the common types of disruptive behavior and present the finding in Table 4. We can see that, as a generic class of problems, DAB has diverse patterns. While a few aspect of DAB is studied in literature, *e.g.*, wakelock leak [52], improper prefetching [67, 55], others are under-explored, *e.g.*, aggressive network retry, excessive storage use, and noisy notifications. The summarized patterns provide us with clues about the changes in mobile OS we need to make in order to restrict common DAB. The reproduced DAB issues will be used as evaluation subjects later.

Disruptive behavior	Cases*
Too frequent connection, aggressive retry	44 (10)
Excessive data transfer over cellular network	42 (10)
Excessive storage, cache	33 (5)
Excessive or stuck notifications	33 (9)
Very high CPU usage	31 (8)
Wakelock leak or overuse	25 (13)
GPS leak or overuse	19 (11)
Too aggressive sensor/GPS updates	15 (6)
Misc. resource leak or overuse	15 (3)
Too frequent wake-up alarms	12 (8)
Sensors leak or overuse	12 (8)
Too frequent broadcasts, receivers	6 (5)
Total	287 (96)

Table 4: Common DAB summarized from our study of 287 real-world cases. *: parenthesized numbers are the cases we successfully reproduced.

2.3 Observations

In addition to the common disruptive behavior patterns, we also make the following observations:

(1). Even expert developers can make mistakes. The misbehaving apps we study include apps written by novice developers or developers who write apps as a hobby (*e.g.* at night after work), as well as apps written by professional teams (*e.g.*, Facebook, Google, Spotify).

Implication 1: *Only relying on developers to always ensure app quality is challenging. The OS needs to anticipate common disruptive app behavior at runtime.*

(2). The root causes of DAB are diverse, including bugs (*e.g.*, duplicate downloads [12]), correct but unfriendly implementation (*e.g.*, fetch all messages from all friends at once [1]), and improper settings (*e.g.*, aggressive update frequency [17]). This is the case even for the same DAB. Take "excessive cellular data transfer" as an example. It could occur *a).* because the app does not support WiFi-only feature; *b).* because the app implements this feature but the default configuration is incorrect: *e.g.*, *"I just installed Podax on my new phone and I forgot to change the setting and it used up nearly my entire month of bandwidth in an hour"* [13]; *c).* because of a complex bug that causes a loop of network activities [21].

This poses challenges for existing tools to statically detect DAB before app release. For example, unfriendly implementation and improper settings that lead to DAB are under-explored in existing literature. In our study, 34% of the cases are caused by improper settings.

Implication 2: *Enhancing OS to deal with DAB dynamically from app execution has the advantage that the resolution does not depend on the root causes of DAB.*

(3). Some DAB is triggered under complex conditions. In 57 cases, the DAB is only manifested under certain external conditions such as when GPS signal is poor, network is unstable, contact-list is large, or remote host does not support password authentication. These defects can escape testing and occur on users' mobile devices when the apps are deployed.

Implication 3: *To protect users from being affected by these complex DAB requires dynamic solutions that run continuously at users' side.*

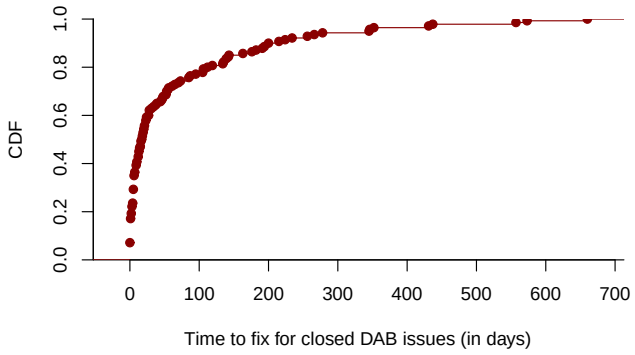

Figure 3: CDF of time from DAB being reported to issue being fixed.

(4). While frustrated by the DAB, users often still need to use the apps for functionalities. For example, a user comments on a battery drain issue caused by excessive wakelock in CSipSimple [10]: *"this issue makes this program less acceptable to me than Sipdroid. Otherwise, I am happy with the UI and satisfied with stability."* Quantitatively, despite having defects, the studied apps on average have overall rating (across all versions) of 4.1 out of 5.

Implication 4: *In dealing with DAB, simple coarse-grained solutions like killing or uninstalling the app are undesirable; the solution should restrict disruptive behavior without sacrificing app main functionality.*

(5). Fixing DAB takes time and/or is not easy. In some cases, it took a long time for developers to fix the defect because developers could not find the defect easily [14] or because the developers do not have enough time to implement the fix [20]. Figure 3 shows the CDF of time from a DAB is reported to it is fixed. The median fix time is 17.5 days and the mean is 70 days. Moreover, 18% of our studied cases still do not have a fix to date (not shown in Figure 3).

Implication 5.1: *While developers are working on fixing DAB, it is desirable for OS to limit the impact of DAB at user side.*

A few apps implement advanced fix such as adding a dynamic mechanism to decide whether to perform an expensive operation or not based on the system health. For example, an app adds a mode in which it will not perform lookups or show pop-ups if the battery is low [18]. However, such dynamic optimization is not easy to implement correctly for many app developers.

Implication 5.2: *Having OS provide runtime mitigation to DAB may help reduce efforts of complex optimization for inexperienced developers.*

3. DefDroid DESIGN

DefDroid aims to restrain DAB without breaking app main functionality or introducing much overhead. The key idea of DefDroid is to execute various fine-grained defensive[2] operations to mitigate app misbehavior, and optionally revert these operations after misbehavior subsides.

3.1 Overview

We make several design decisions guided by our study (§2). First, DefDroid only applies coarse-grained actions such as killing

[2]In the sense that the OS is vigilant about potential DAB and protects the ecosystem/users from the negative impact of DAB.

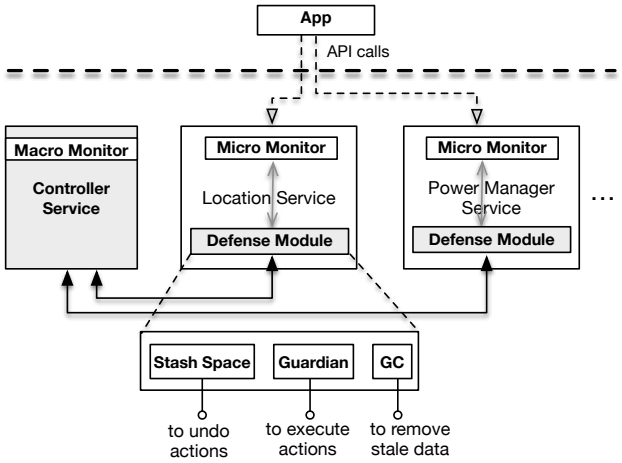

Figure 4: Overview of DefDroid

apps as a last resort. This is because the app misbehaviors we address are mainly due to programming mistakes instead of malice, and users need to use the app's main functionalities. Second, DefDroid only corrects specific apps instead of the whole system. It avoids employing actions that have global impact such as stopping sensors, underclocking CPU or adjusting global timer. Third, to avoid hurting user experience, DefDroid currently does *not* execute defensive actions to system apps or foreground apps that users are interacting with because aggressive behavior during user interaction is often expected by users.

There are several challenges in designing DefDroid: *a).* handle a diverse and growing set of app misbehaviors; *b).* collect fine-grained app activity information to monitor misbehavior without significant overhead; *c).* execute actions precisely to misbehaving app and responsible requests; *d).* determine appropriate policies for the defensive mechanisms, and *e).* revoke the imposed adjustments when the misbehavior subsides.

Figure 4 shows the architecture of DefDroid. DefDroid adopts a modular design. It uses a *controller service* to monitor macro-level system conditions (*e.g.*, battery, storage), and to manage an extensible set of *defense modules* to address different DAB. A defense module resides in an OS subsystem together with a *micro monitor* who records important app activities. Within a defense module are: 1). a *guardian* that analyzes monitor information and executes defensive actions to potential DAB; 2). a *stash space* to store information necessary for undoing defensive actions; and 3). a *garbage collector* to remove stale monitor and stash data to control overhead.

DefDroid has a notion of *permissive* and *defensive* modes. In the permissive mode, DefDroid provides services faithfully to apps like what its base OS does. When there are signs of unhealthy system conditions and suspicious app activities (*e.g.*, frequent wake-ups, lingering wakelock), DefDroid may transit into the defensive mode. In the defensive mode, DefDroid executes fine-grained defensive actions to misbehaving apps, *e.g.*, decreasing an app's timer frequency temporarily, releasing long-held wakelock.

3.2 Controller Service

DefDroid adds a controller service to its base OS to record macro-level system conditions such as battery level and data usage at a regular interval. But the macro information alone is not enough to infer whether an app is misbehaving. Nor is the information sufficient to devise a fine-grained action.

```
protected class AlarmMonitor extends DefenseMonitor {
  private AlarmTable<PendingIntent> mTable;
  public void alarmSet(Alarm alarm) {
    mTable.setDescriptor(alarm.operation, alarm);
    mTable.updateAlloc(alarm.operation);
  }
  public void alarmCancelled(PendingIntent operation) {
    mTable.updateRelease(operation);
  }
  public void alarmTriggered(PendingIntent operation,
      long duration) {
    mTable.incrementUsage(operation, duration);
  }
}
```

Figure 5: A micro monitor inserted in `AlarmManagerService`. We modify implementations of the `AlarmManagerService` API methods such as `set` and `cancel` to invoke the micro monitor.

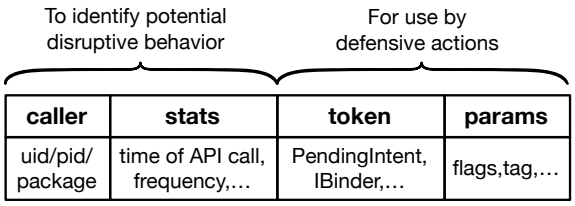

Figure 6: Typical data entries in a micro monitor to record app API request information.

Therefore, DefDroid delegates the main mechanisms of addressing DAB to a set of specific defense modules that live inside various important subsystems. The controller is lightweight to facilitate these modules. For example, the controller propagates macro-level system conditions to the defense modules and manages the life cycle of these modules. The controller also exposes the configurations, such as the policies of defensive actions, and a whitelist of apps, to end users.

3.3 Micro Monitors

To enable fine-grained defensive mechanisms, DefDroid monitors not only system-level conditions but also app-level activities. This is accomplished by inserting a set of micro monitors in major OS subsystems that apps interact with, *e.g.*, the power service and the location service. Figure 5 shows a snippet of the micro monitor class for the `AlarmManagerService`.

The exact information recorded by a micro monitor is component-dependant. But it usually covers app API requests information, including four parts (Figure 6): the caller (UID or package name), request stats (*e.g.*, first time a type of request is made, request rate), tokens that uniquely identify requests (*e.g.*, `IBinder` object in Android) and request parameters.

We need to track and aggregate app request stats because judging from an individual API request alone usually cannot distinguish between legitimate requests and disruptive requests at the time of the API call. For example, a timer request with a small interval may not adversely impact the system if it is used only for a brief period. To identify frequent wake-up misbehavior, DefDroid needs to track the timer triggering counters (Figure 5). As another example, a wakelock `acquire` call by an misbehaving app is no different than the calls from other apps. It is only after some time, well-behaved apps call `release` while the misbehaving apps forget to call `release`. Therefore, DefDroid needs to track the wakelock holding time to identify potential wakelock leaks.

Recording tokens and request parameters in micro monitor is

Action type	Example
Release	release long-held wakelock, GPS
Slowdown	enlarge alarm interval, reduce sensor frequency
Delay	delay excessive cellular network requests
Approximate	use coarse-grained location
Downgrade	switch to passive GPS provider
Deprioritize	decrease scheduling priority
Block	reject frequent alarm requests for a while
Warn	warn users about large data consumption
Kill	kill naughty app processes or services (last resort)

Table 5: Types of defensive actions

needed because DefDroid may take defensive actions such as decreasing frequency of aggressive alarms. These defensive actions need to know the details of the original request. Tokens like `IBinder` objects or resource descriptors are the unique keys to identify the original requests. Together with the original request parameters, DefDroid can tweak an app's behavior.

Some of the OS subsystems that micro monitors reside in already store some request information internally as part of API implementations (*e.g.*, the alarm object in Figure 5). In such situation, the micro monitor simply keeps references in its data structures and only needs to update request statistics. For information not stored originally, the micro monitor creates data structures to track the information.

To control the monitoring overhead, micro monitors do not track all app activities. Only the kinds of requests that may cause adverse impact to the systems will be monitored. Moreover, most data in micro monitors will be removed when apps make requests like `release` and `cancel`. Each defense module also has a garbage collector that periodically cleans up old monitor data to control the memory overhead (Figure 4).

3.4 Guardians

Guardians in DefDroid are the main components to take fine-grained actions to reduce various disruptive behaviors. They periodically check the monitor data for potential DAB patterns and carry out actions. A guardian will register with the controller service when its residing subsystem starts. The guardian gets commands from controller, *e.g.*, to start/stop. It also obtains updates on system-level conditions and policies from the controller.

3.4.1 Defensive Action

Defensive actions are the key operations to tackle DAB at a fine granularity. We design the defensive actions by devising a counter-action for each DAB pattern summarized from our study (Table 4). In executing the defensive actions, DefDroid leverages the caller, requests and token information from micro monitors to precisely target the responsible app and requests.

Table 5 shows the types of defensive actions used in DefDroid. These actions may release resources, adjust frequencies, trade off accuracies, change priorities, delay/deny requests, or warn users. For example, a guardian may release an expensive resource if the requesting app has not been active for a while to prevent the resource-leak disruptive behavior. For the aggressive alarm behavior pattern, a guardian may increase the timer interval. For overly frequent wakelock acquisitions, a guardian may block future wakelock acquire requests from that app for a while. For aggressive GPS requests, a guardian may opt to return coarse-grained location information, switch to passive location providers (*i.e.*, receiving location

```
public class RequestDelayerBase<T> {
  public DelayResult delayTime(String owner, T request,
      IRequestReviver<T> reviver, long time, ...) {
    DelayData data = mDelayMap.get(owner);
    if (data == null || !data.active)
      return DelayResult.NOT_IN_LIST;
    if (data.requestQueue.size() > MAX_BUFFER)
      return DelayResult.BUFFER_FULL;
    data.requestQueue.add(request);
    handler.postDelayed(new Runnable() {
        public void run() {
          reviver.reviveRequest(request);
          ...
        }
      }, time);
    return DelayResult.DELAYED;
  }
}
```

Figure 7: Outline of a basic `delay` by time action.

updates only when there are other apps requesting them) or unregister the app's listener.

Like micro monitors, implementing these defensive actions depends on the detailed internals of the specific subsystem: *e.g.* to implement *release* action for wakelock, we periodically check a table of request information such as start time for all wakelocks, and remove the corresponding lock object from an internal array of wakelocks when the duration exceeds a pre-defined threshold[3]; to implement *slowdown* for alarm, we adjust the `repeatInterval` field of an internal data structure; to implement *delay* by time action, we put the request in a buffer and process the request after certain time (Figure 7).

The defensive actions are mainly designed based on what a well-behaved app would do for the benefit of the ecosystem. For example, a well-behaved app would do exponential back-offs when it fails to connect to network, similar to our *slowdown* action; a well-behaved app would only use high-accuracy location when needed and would otherwise switch to lower-accuracy or lower-update-frequency location [11], similar to our *approximate* or *slowdown* action.

When multiple defensive actions are applicable for a DAB, DefDroid tries the actions with least perturbation first before executing more strict actions. For example, when an app uses sensor aggressively in background for a while, DefDroid may first adjust the sampling period or max report latency in the app's sensor request if possible; if it is ineffective, DefDroid may decrease the thread priority or release the listener. A defensive action may also be applied multiple times. For example, for frequent wake-up behavior, the guardian in alarm service will increase the alarm request interval. If it is ineffective, it will keep increasing the interval for multiple times.

3.4.2 Policy

The guardians in DefDroid are controlled by a number of settings. These settings define the thresholds of disruptive behavior listed in Table 4, *e.g.*, wakelock session duration that is considered "too long", number of updates in a period that is considered "too aggressive".

In studying real-world DAB issues (§2), we find disruptive behaviors often clearly distinguish from normal behaviors in terms of request patterns. This makes choosing the settings relatively easy and less sensitive to tuning. For example, Figure 8 shows the CDF

[3]Wakelock object is different from traditional locks: it is only a kernel reference counter to hardware components.

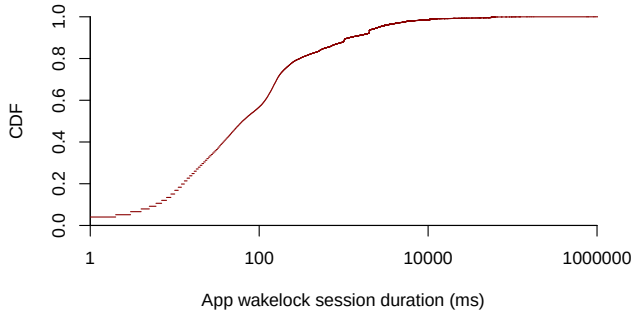

Figure 8: CDF of 46,875 wakelock request sessions (from `acquire` to `release`) from 36 popular apps over a one-week period on a Samsung Galaxy S4 device.

Parameter	Setting
Rate limit minimum window	3 min.
Wakelock duration throttle	5 min.
Wakelock alloc. rate limit	3 per min.
GPS duration throttle	10 min.
Sensor alloc. rate limit	20 per sec
Network checker frequency	3 min.
Data threshold per time slice	5 MB
Max consecutive overused slices	4
Back-off multiplier	1.5

Table 6: Some default settings for DefDroid.

of wakelock request sessions gathered from 36 popular apps that use wakelock over a one-week period in a Samsung Galaxy S4 device. We can see the majority of wakelock request session is very short (the 99th percentile is 12 secs) with a few clear outliers (the long tail).

To set the thresholds for disruptive behavior, we empirically profile many popular apps for a long period offline to obtain the distribution of some interested metrics. Then we choose an empirical setting that significantly deviates from the majority distribution.

For some metrics such as mobile data consumption, simply using a single threshold may falsely alarm some legitimate bursty app activities. To avoid this problem, we divide the app behavior monitoring into time slices (*e.g.*, 3-minute slice), and mark a time slice as bad when a threshold is crossed (*e.g.*, used 5+ MB data). Then we only consider that app is misbehaving when a number of *consecutive* time slices are all bad. Table 6 shows some default settings in DefDroid.

The defense policies also specify the intervals to check DAB patterns, the order to try different actions, and parameters related to a specific defensive action (*e.g.*, the extent to increase a timer interval). By default, system apps are exempt from the defensive mechanisms to protect the most important functionalities like phone call and messaging. Additionally, in a given checking interval, guardians currently will not apply defensive actions on apps that are in the foreground to avoid affecting user experiences. But the monitors still record important requests in case defensive actions need them later. When the device is being charged and connected to WiFi, DefDroid does not enter defensive mode because many apps are designed to perform heavy and aggressive work in this scenario and the impact usually does not bother users.

DefDroid also provides user interfaces to show apps being penalized by defensive actions, and to allow users to configure a white-

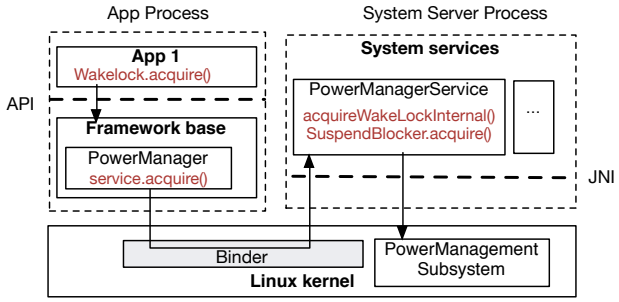

Figure 9: Several layers that a typical Android API call goes through.

list of apps to be immune from penalty. For potentially intrusive action such as releasing lingering GPS listener, DefDroid asks users to confirm before executing the action.

3.4.3 Reverting Defensive Actions

The defensive actions executed by DefDroid are usually temporary. After an observation period, if the actions are effective or healthy external conditions occur (*e.g.*, battery recharged, connected to Wi-Fi), the actions might be reverted. For example, if the guardian adjusted the timer frequency for an app, it may later restore timer frequency for that app. Reversion gives chances for misbehaving apps to behave normally again.

To implement reversions, the defense module reserves a stash area to bookkeep recently applied defensive actions (Figure 4). The stash area also keeps important tokens and parameters for reversions. The data in the stash area is removed when apps explicitly revoke the recorded requests, *e.g.*, canceling timer, terminating unexpectedly. The garbage collector in a defense module will also clean the data in stash area after it expires.

4. IMPLEMENTATION

We have built DefDroid in Android AOSP 4.4.3, AOSP 5.1.1 and Cyanogenmod 11. We implemented 10 defense modules (namely Wakelock, Location, Sensor, Alarm, Network, Notification, CPU, Storage, Bluetooth, and Light). These modules are managed by a controller service added to the Android system server [66].

Layers to implement the defense modules. There are several layers in the Android stack (*cf.* Figure 9) a defense module could reside in: the Android framework base layer in app address space (*e.g.*, `PowerManager` class), the system service in system server (*e.g.*, `PowerManagerService` class), and native Linux kernel (*e.g.*, power management subsystem).

While placing the defense modules at the framework base layer allows direct access to app objects, doing so can incur significant overhead (*e.g.*, monitoring data structures) because the modules live in *every* app's address space. This is a lesson we learned during our initial prototype of DefDroid in this way with the Xposed framework [26]. Moreover, directly manipulating app objects can cause side effects to app state. Our initial Xposed-based prototype occasionally causes apps to crash.

Therefore, we move away from this approach and implement the defense modules mainly at the system service layer. One challenge of doing so is that DefDroid is unable to directly access app objects to implement defensive actions. We solve this issue by leveraging the IPC tokens (mainly `IBinder`) that app uses to communicate with system services. With the token, the defensive actions can uniquely identify a request, and invoke the IPC methods for an API

call within the system service as if the request comes from the app. In this way, the defensive actions will not mutate an app's state.

The sensor and camera defensive modules currently reside in the framework layer because there are no corresponding system services in the original Android. We also modify the native code layer, *e.g.*, the scheduler and network daemon, to support low-level defensive actions.

Avoid breaking invariants in OS subsystems. Since defensive guardians live inside subsystems, some defensive actions may alter the system internal states. Incautious modifications can break the original invariants in the subsystem. For example, releasing an app's wakelock in Android is essentially removing the `IBinder` object from an internal array in the power management subsystem. But if the guardian only does the removal, the internal state of the subsystem may be incorrect, because when an app calls `release`, the subsystem will also update other internal data (*e.g.*, battery statistics). To address this issue, instead of directly manipulating system states, the defensive guardians in DefDroid often implements an action by making API calls as if the call is from an app. For example, when a guardian alters a timer frequency, it does so by calling `cancel` API and then `set` API with the updated parameters.

Orders of starting defensive modules. Some defensive modules reside in the first few core subsystems but they have dependencies on subsystems that are started later. Running these modules immediately when their residing subsystems start may cause exceptions. We address this issue by starting the defensive service as the last system component. Since defensive modules will register with the defensive service and can only be started by the defensive service, when a defensive module is started, its dependent subsystems will be running.

5. EVALUATION

We first evaluate DefDroid on the 96 real-world DAB cases that we reproduced in §2. We measure the effectiveness of DefDroid in curbing DAB, as well as quantitative benefits (e.g., savings of battery and cellular data).

In addition, we evaluate DefDroid on 32 *newly* collected real-world cases that cover all DAB symptoms listed in Table 4. As some of the new cases are caused by different programming mistakes than the studied ones, the evaluation tells whether DefDroid, as a dynamic solution, is effective in handling *new* DAB cases.

Another critical aspect of a defensive OS like DefDroid is its impact on the usability of apps. To understand this, we first quantitatively measure DefDroid's usability impact on the evaluated apps. Second, we conduct a small-scale user trial to observe how DefDroid performs in a user's daily use of smartphone. Third, we deploy DefDroid to 185 *real users* through the PhoneLab [47]. Lastly, we measure the performance overhead of DefDroid compared to vanilla Android.

5.1 Experimental Setup

The experiments are conducted on two Motorola G devices and one Nexus 4 device running DefDroid that is modified from Android 4.4.3. We use the Qualcomm Trepn Profiler 5.1 [19] to profile system and app performance values. Experiments that involve performance values are all run on the Nexus 4 device for consistency. The default defense policies used in the experiments are set empirically as described in §3.4.2, and we evaluate how different policies affect the results in §5.5.

When evaluating the 128 DAB cases, we run each case in turn on DefDroid and compare the effect with running these cases on unmodified Android. For the sake of comparability, we toggle a knob in DefDroid to control the changes that are added to Android. When turned off, the defensive service and all modules will be completely disabled so that the system behaves the same as unmodified Android. In this way, the experiments are conducted within the same ecosystems.

For each case, we first interact with the app for 5 minutes and perform user actions to trigger the misbehavior. Then we return to the Home screen and wait for 25 minutes to observe the effect of disruptive behavior. For fair comparison, we interact with the app in similar way when switching from DefDroid to unmodified Android.

Studied cases		New cases	
Total	**Handled**	**Total**	**Handled**
96	94	32	31

Table 7: The number of real-world DAB cases DefDroid succesfully handles among all the evaluated DAB cases.

Resource	Reduction		
	Min	**Median**	**Max**
Power	72mW, 6%	189mW, 21%	481mW, 87%
Mobile data	17MB, 15%	54MB, 34%	183MB, 86%
Storage	30MB, 25%	48MB, 37%	145MB, 90%

Table 8: Resource consumption reduction value and percentage for running the real-world DAB cases on DefDroid compared to Android.

5.2 Curbing Disruptive App Behavior

5.2.1 *Overall Result*

We first show how effective is DefDroid in handling the 128 real-world DAB cases. A case is considered to be successfully handled when DefDroid took actions on the misbehaviors during the experiments. As shown in Table 7, DefDroid successfully handled 125 out of 128 cases in total (94 studied cases and 31 new cases).

DefDroid misses 3 cases due to the defensive policy settings. In our default policies, we define an app as potentially over-consuming mobile data if it crosses a threshold for 4 *consecutive* monitoring intervals. This policy can tolerate legitimate spike in data usage. But in two of the missed cases, the apps did not cross the threshold for one interval. To catch them requires adjusting the default policy settings. The third missed case is related to storage over-use. The consumption did not exceed the empirical thresholds that we set via profiling popular apps. But the user considered the consumption abnormally high given that the app is text-based. We plan to improve defensive policies by considering app categories and usage history.

5.2.2 *Resource Consumption Reduction*

From a user's perspective, the impact of DAB is mainly perceived as excessive usage of resources like battery and cellular data. A primary benefit of using a defensive mobile OS like DefDroid is to restrict such negative impact. Therefore, for the 125 cases that are handled by DefDroid, we further measure the reduction in resource consumption compared with unmodified Android.

Table 8 shows the reduction results (both absolute amount and percentage-wise) for power, mobile data and storage. For each resource, we measure cases that have impact on that resource. The re-

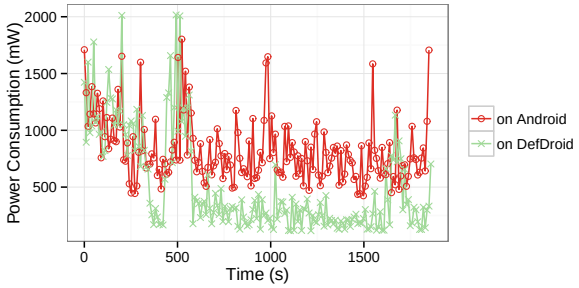

Figure 10: Power consumption comparison for a DAB case running on Android versus running on DefDroid.

Metric	Android	DefDroid
Wakelock held time	26–30min	7–10min
GPS held time	30min	10–13min
Sensor held time	30min	6–8min
Other resource held time	30min	6–9min
Number of wake-ups	61–203	26–49
Number of broadcasts	64–80	28–31
Number of notifications	43–110	13–41
Number of connections	180–1173	64–477
CPU load	70–100%	43–67%

Table 9: Micro metrics comparison of running the DAB cases on Android and running them on DefDroid.

sults show that, the defensive mechanisms in DefDroid drastically limit the impact of DAB and achieve significant resource consumption reduction.

Figure 10 shows an example that illustrates the power consumption of running an evaluated DAB case (OSMTracker GPS leak) on DefDroid versus on unmodified Android. Since we interact with the app on DefDroid similarly as on unmodified Android in the first 5 minutes, the app power consumptions on the two systems are similar in the initial phase. But at around the 10th minute, DefDroid unregistered the leaked listener which still persisted on the unmodified Android. Therefore the power consumptions on the two systems start to diverge.

5.2.3 Micro Analysis

In addition to the resource consumption reductions, we also analyze key system micro metrics that reflect the symptoms of a given DAB pattern, *e.g.* wakelock held time for the wakelock leak/overuse pattern. For each evaluated case, we compare the micro metrics on DefDroid with the metrics on unmodified Android.

Table 9 shows the result. Since there are multiple relevant cases for a particular metric, we show the values in a range form. The comparison results are in line with macro resource consumption analysis: the defensive mechanisms in DefDroid successfully restrained the effect of evaluated misbehaviors at the micro level.

Moreover, we analyze the defense actions that DefDroid took when handling the evaluated DAB cases. In total, 258 actions were executed during the experiments. In dealing with 89 cases, DefDroid applied multiple actions (*e.g.*, *approximate* then *release*) or one action multiple times (*e.g.*, *slowdown*).

Figure 11 shows the distribution of the defensive actions applied by DefDroid. The most applied actions are *slowdown*, *block* and *release*. Interestingly, we observe that having a diverse set of defensive modules in DefDroid helps catch some misbehaviors more quickly. For instance, in dealing with some "endless connection" misbehavior, the CPU and alarm defensive module took actions (as they used CPU or alarms too aggressively) to limit the disruptive behavior before the network defensive module took actions. Figure 11 also shows that the *kill* actions were not invoked during the experiments.

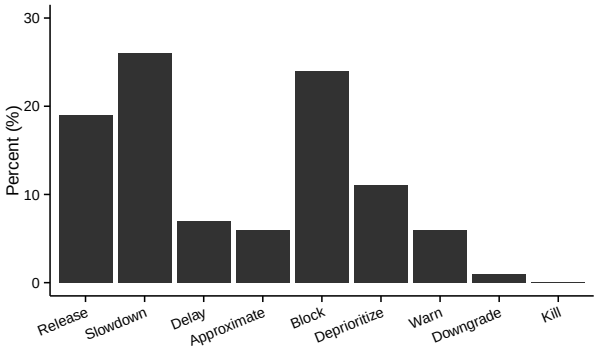

Figure 11: The distribution of defensive actions that DefDroid took during experiment. Refer to Table 5 for examples of a defensive action.

Impact	Cases
No side effect	40 (32%)
Deferred executions	49 (39%)
Failed requests	31 (25%)
Checked exceptions*	5 (4%)
Termination	0 (0%)

Table 10: Impact of DefDroid defensive mechanisms to evaluated disruptive apps. *: exceptions that compilers enforce developers to handle, *e.g.* IOException.

5.3 Usability Impact

Being effective in curbing DAB is not enough. Another important requirement of DefDroid is to still preserve the main functionalities of the misbehaving apps.

To meet this requirement, when DefDroid detects a potential misbehavior, it tries defensive actions with least perturbation to app usability, before executing more strict actions. Moreover, when executing an action such as adjusting an app's frequent alarms, DefDroid makes gradual adjustments instead of abrupt changes. DefDroid also exempts foreground apps from defensive actions to allow legitimate heavy usage (*e.g.*, using Google Maps for a long time will not be affected by DefDroid).

We analyze the usability impact of our defensive actions to the misbehaving apps used in the experiments from both execution logs and theory of related OS mechanism. Take the wakelock as an example. A wakelock is essentially a kernel reference counter to certain hardware components (typically CPU). As long as *any* app has an active wakelock, the hardware components will not go into low-power state. Apps use wakelock to ensure the hardware component stays on to finish tasks that are deemed important. When an app holds a wakelock for too long, DefDroid may force a *release* on that wakelock. From app perspective, the effects of the *release* action depend on several situations:

a). The app leaks the wakelock and the tasks have already finished. Then the *release* action curbs the misbehavior and causes no side effect to the app at all.

b). The app leaks the wakelock and the tasks have not finished yet. The *release* action may also induce no side effect. This is because as long as there are some other apps holding an active wakelock at that moment, the hardware component will keep on, giving chance for the affected app to finish the tasks as usual. If the released wakelock is the last reference, the CPU will be put into low-power state soon after *release*. The system will freeze the app state before suspending. When important interrupts wake up the system later, the app state will be resumed. Therefore, the *release* action causes delay in app execution. In certain scenarios, when the app resumes execution, it might cause exceptions (*e.g.*, `IOException`). These exceptions are usually enforced to be handled by the developers, *i.e.*, the corresponding exception handling code in the app will be triggered.

c). The app does not leak the wakelock but holds it for unusually long. The *release* action does not affect the app to later refer to and release the lock object. This is because the defensive action is executed inside system services; Therefore, depending on whether the tasks have finished or not, the impact is similar to *a)* or *b)*, respectively.

The analysis result for usability impact to all the evaluated apps is presented in Table 10. It shows that the executed defensive actions cause only marginal usability impact for most of the evaluated apps. For a significant percentage of the cases, the defensive actions do not cause any side effects to the misbehaving apps. Moreover, DefDroid does not cause crashes for the evaluated apps.

The significant percentage (32%) of no-side-effect cases is mainly attributable to our designs. Additionally, some defensive action (*e.g.*, *release*) is in fact a "hot fix" for certain DAB cases like resource leak. Moreover, our choice to implement the defensive modules in system server rather than app address space also ensures that the action does not tamper with app internal state. Lastly, from the app's perspective, the effect of some defensive actions is the same as an environment condition that the app has to cope with, *e.g.*, network disconnection and losing GPS signal.

In 80 cases, the defensive actions cause deferred execution or failed requests. Such impact may be undesirable for well-behaved apps. But note that the evaluated subjects are apps that are behaving aggressively, *e.g.*, a weather app updating locations every minute, a chat app entering infinite connection loops due to a bug. DefDroid makes these behaviors less aggressive but still maintains the main functionalities of these apps, *e.g.*, reduce the weather location update frequency to be every three minutes. Such impact is usually not noticeable to users as validated in §5.4.

In the remaining 5 cases, the defensive actions caused checked exceptions – exceptions that the compilers enforce developers to handle, *e.g.*, `SSLException` due to connection time out when the app resumes. Therefore, the defensive actions triggered the error handling code to be executed as if external conditions (*e.g.*, network disconnection) occurred. These exceptions did not cause the apps to crash or enter unexpected states.

5.4 Real User Deployment

Except for evaluating DefDroid on existing real-world DAB cases, we further understand how DefDroid performs in a real user's daily usage scenario.

We first conduct a trial with 4 volunteer users for 10 days. We flash the DefDroid system image to these users' smartphones and ask the users to report any suspicious problems encountered during their daily use. At the end of the trial, we collect the DefDroid logs from their devices and interview the users for their experiences.

App	Disruptive behavior
cClock	Constrant GPS search
The Economist	Wakelock leak or overuse
Sina News	Long-held GPS
Xiami	Excessive wakelock requests
Kik	Too frequent wake-ups
Zillow	Excessive toasts (notifications)

Table 11: Newly found DAB issues that DefDroid identified and handled in user trials.

In total, we received two reports of DefDroid falsely warning the users about potential large mobile data usage. Both users think the data usage is acceptable, and they simply chose the default ignore action provided by DefDroid. Beside these two, we did not receive any other reports during the trial; nor are any other noticeable anomalies/slowdowns reported in the final interview.

Interestingly, when we analyze the collected logs, we find DefDroid took a number of actions to several apps in the user trials. We further confirmed these misbehaviors DefDroid identified by reproducing them in our smartphones. Table 11 shows these new DAB issues. Some users are surprised about these findings. One user said, *"I installed the Xiami app but found it did not support the U.S. users, so I did not use it at all afterwards."* But DefDroid discovered the app was requesting wakelock excessively in the background and denied its requests for certain time. We reproduced the misbehavior and found it could drain battery quickly. With DefDroid, the users did not experience negative impact such as fast battery drain for these new DAB issues during the trial. Some of the new DAB issues are also confirmed by other users or developers, *e.g.* the issue of cClock constantly searching GPS and draining battery [9, 8] and the issue of Kik battery drain due to excessive wake-ups.

In addition to the small-scale user trial, we have also deployed DefDroid via the PhoneLab testbed [47] to 185 real users. The deployment follows PhoneLab release policy: the platform change is first rolled out to a small group of PhoneLab developers to be used for at least one week to make sure the change does not cause significant user experience disruption; then the change will be pushed to all users.

The DefDroid experiment on PhoneLab lasted for 43 days from 2015/09/21 to 2015/11/3. During this period, DefDroid applied actions on 81 unique apps in 105 of the 185 participants. We recreated the logged disruptive behavior for 57 apps such as Viber, TWC WiFi Finder, Bejeweled Blitz, GPS Odometer. Some cases were initially puzzling to us based on the collected logs. For example, we found DefDroid applied actions on the NY Times app in several devices because the app used sensor aggressively even when users were not using it. We decompiled the app binary to make sense of the behavior. Even though the source code is obfuscated, based on the path name of the source file that made the sensor requests, we suspect the aggressive behavior came from an advertisement module that the app is using. There are 6 *potential* false alarms (from sleep tracking apps and fitness apps). These aggressive behaviors are likely expected by users. For those logged disruptive behavior that we could not reproduce, we suspect it is because the latest version fixed the issue or the user conducted some special interactions.

Through the PhoneLab experiment, we also learned some lessons: *1).* Our primary goal of the PhoneLab deployment is to get an as-

	CPU Load	Memory	Power
Android	37.5%	1721.9MB	1688.4mW
DefDroid	39.2%	1749.8MB	1719.8mW
Overhead	1.7±0.3%	27.9±0.9MB	31.4±7.4mW

Table 12: Performance overhead of DefDroid with 95% confidence interval.

sessment on whether DefDroid may break app functionality in real world. In this regard, the experiment shows positive result: for all the apps that DefDroid applied actions during the experiment, we did not receive any user complaint that the functionality of these apps became broken or eccentric. But we got some user complaints about overall battery drains. Before the deployment, the authors had already used phones running DefDroid based on Android 4.4 for more than one month without encountering major issues. When we were requesting to deploy DefDroid on PhoneLab, the PhoneLab codebase was in the middle of migrating to Android 5.1. Due to release time constraint, we ported DefDroid from Android 4.4 to 5.1 in about 10 days. As a result, even though we preliminarily tested the new DefDroid, we did not expose the inefficiency under Android 5.1 that caused battery drain for some users. We later found and fixed the issue; *2).* When we started to analyze the logs after deployment, we realized the deployment is also a good opportunity to collaboratively tune the defense policies (*e.g.*, sensor rate limit). The log statements we added in DefDroid for the experiment missed some important details that could have been benefitial to understand about DAB in the wild.

5.5 Different Defense Policies

The effectiveness of DefDroid depends on both the defensive actions and policies. We set the default policies by profiling the behaviors of many popular apps and empirically choose the settings. Besides the default policies, DefDroid provides users with two additional policy templates, a "conservative" policy template and a more "aggressive" one, to trade-off the defense and the impact. Under the conservative policies (*e.g.*, GPS duration throttle increased to 20 min), DefDroid may miss some DAB but the usability side effects tend to be reduced. Under the aggressive policies, DefDroid may catch more DAB but at the cost of more side effects.

We experiment DefDroid with both the conservative policies and aggressive policies on the 128 DAB cases. Compared to the default policy, the conservative policies would miss 6 more cases while increasing the no-impact cases in Table 10 by 11; the aggressive policy would catch all 3 missed cases in default policy while decreasing the no-impact cases in Table 10 by 8.

5.6 Performance Overhead

We evaluate the performance overhead of DefDroid during normal usage scenario. We repeatedly interact with 10 popular apps such as Facebook, Angry Birds, and Gmail in turns for a total session of 30 minutes. During the interaction session, these apps will switch between background and foreground. The experiments are first conducted three times on our Nexus 4 device running DefDroid, and then conducted three times on our Nexus 4 device running unmodified Android. In all the experiments, we try to interact with the apps in the same way for fair comparison. Even if DefDroid does not execute defensive action during the experiment session, DefDroid still needs to actively monitor app activities, which is the main component of DefDroid's overhead.

We measure the CPU load, memory usage and power consumption of DefDroid and unmodified Android when running the above workloads. Table 12 shows the averaged results and overhead with 95% confidence intervals. The results indicate that DefDroid incurs little overhead compared with unmodified Android.

6. DISCUSSION AND LIMITATIONS

API semantics guarantees to apps: A design trade-off that needs to be considered in making mobile OS more defensive is the API guarantees to apps. Some defensive actions may relax the original API contract under aggressive condition. For example, in DefDroid, an app's wake-up requests may be suppressed by the OS if they are too frequent. In contrast, vanilla Android always satisfies the app requests. We believe that the pervasive DAB issues and the rise of novice app developers make such trade-off necessary to prevent DAB from impacting users. As mentioned in §1.3, this trade-off is echoed by some recent changes in mainstream mobile OSes. For example, all repeating alarms are inexact starting from Android 4.4. DefDroid leverages fine-grained action and reversion to minimize the compromise on API contract. To app developers, such nondeterminism might provide incentive for more cautious usage of APIs (since nondeterminism only arises when the request pattern is potentially aggressive). In the future, we plan to explore exposing some callbacks to developers when a contract is going to be broken to give apps some grace period to react.

Other mobile OSes: Many of the DAB issues we study such as aggressive sensor usage, frequent wake-ups, and excessive mobile data usage are not specific to Android but widely exist in other platforms including iOS and Windows Phone as well [49, 42, 48]. We choose to implement DefDroid on Android because it is open-sourced. We leave studying and addressing the DAB issues in the other platforms for future work.

Using DefDroid as a developer tool: The main goal of DefDroid is to provide a last level of runtime defense to prevent DAB issues from affecting users. But it is also possible to use DefDroid as a developer tool to help developers catch their apps' disruptive behavior that is hard to be exposed with testing or static code analysis. For example, developers can run their apps in DefDroid for a while before a release, and then use the action logs from DefDroid to determine if potential DAB defects exist in the new version.

Limitations of DefDroid: While our evaluation shows DefDroid is effective on real-world DAB, there are several limitations that we plan to address as our future work. First, each defensive module and action in DefDroid is implemented separately because different resources are managed differently in the base OS. We are looking into whether designing a more generic approach of managing different resources would allow the techniques in DefDroid to be more systematic. Second, since DefDroid does not understand the semantics of app configurations for users, the defensive actions may overrule user configurations. For example, a news app may aggressively refresh content because users configure it so. Users currently can white-list the app to preserve the behavior. Third, the policies for defensive actions in DefDroid are statically set based on offline profiling. We plan to adjust the policies dynamically based on app usage history. Fourth, there may be new DAB patterns that cannot be addressed by the current defensive mechanisms in DefDroid. But DefDroid makes it easy to add new defense modules.

7. RELATED WORK

A few aspects of DAB have been explored individually in literature, *e.g.*, energy bugs [49, 41, 67, 52, 65], inefficient network activity [59, 55, 33, 39, 40, 53, 30], abnormal battery drain [35, 44,

48, 50, 34]. Our work builds upon these prior work but uniquely investigates the generic DAB problem as a whole, and proposes solution from mobile system perspective to address DAB. DefDroid enhances mobile OS to react to a wide range of DAB issues at runtime.

Mobile OSes allow apps to run in background to perform certain task. But background task is often responsible for abnormal battery drain [36]. Many task managers exist to allow users to kill heavy background tasks. ZapDroid [62] automatically disables infrequently used apps. These coarse-grained solutions break app usability and suffer from well-known issues [24, 15]. Amplify [2] and TAMER [45] uses Xposed framework [26] to dynamically hook app method calls in app address space and release/throttle wakelock and GPS requests in background tasks to save battery. While our work takes similar runtime mitigation approach for the two issues, we study a wide range of DAB issues that are not addressed by these tools. Our solution lies natively in the OS to systematically address DAB. Additionally, we comprehensively evaluate the app usability impact of runtime mitigation, which is not investigated in the prior work.

Compared to work that optimizes mobile operating systems such as Ghost hint [29], Cinder [60], Rio [28], K2 [43], and Android Doze mode [16], we focus on improving mobile OS to consider inexperienced app developers who can make various programming mistakes, and proactively protect users from DAB.

The idea of defensive actions in DefDroid is inspired by failure-oblivious computing [58, 54, 31]. But rather than trying to tolerate failures like memory errors of server applications to enhance availability, DefDroid aims to limit the negative impact of DAB at runtime.

8. CONCLUSION

As the mobile developer population and app market continue to soar, disruptive app behavior is an emerging class of issues that calls for more system-level solutions. In this paper, we study a large number of real-world DAB issues and present DefDroid, an Android-based OS that is designed to be more defensive and use fine-grained defensive mechanisms to curb DAB dynamically.

Our evaluation of DefDroid on 128 real-world DAB cases shows that, with careful designs and implementation, a more defensive mobile OS can not only curb DAB effectively but also achieve the effectiveness with little performance overhead and marginal negative usability impact to apps.

DefDroid found and addressed 6 new DAB cases in the field during user trials. We have deployed DefDroid to 185 *real users* via the PhoneLab testbed for 43 days and found DAB issues from at least 57 apps. DefDroid is available at http://defdroid.org.

9. ACKNOWLEDGEMENTS

We thank the MobiSys reviewers and our shepherd, Lenin Ravindranath, for their valuable feedback that improved the paper. We appreciate the great help from the PhoneLab team, especially Jinghao Shi and Geoffrey Challen, on deploying DefDroid on PhoneLab. We are very grateful to the volunteer users and the PhoneLab participants for using DefDroid, and providing useful feedback for us to improve DefDroid. This work was supported by NSF CNS-1017784 and CNS-1321006.

10. REFERENCES

[1] Aggressive behavior in AndStatus - massive requests of "users/show". https://github.com/owncloud/android/issues/554.

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

[3] Android Central. www.androidcentral.com.

[4] Android Forums. http://androidforums.com.

[5] Android WakeLock. http://developer.android.com/reference/android/os/PowerManager.WakeLock.html.

[6] App BetterWeather high battery drain in environment with poor GPS signal. https://github.com/MarcDufresne/BetterWeather/issues/6.

[7] Battery drain because of long-lived wakelock in Kontalk. https://github.com/kontalk/androidclient/issues/143.

[8] cClock app bug report. https://jira.cyanogenmod.org/browse/CYAN-2742.

[9] Clock is draining battery due to constant GPS search. http://forum.cyanogenmod.org/topic/80799-clock-is-draining-battery-due-to-constant-gps-search/.

[10] CSipSimple battery consumption issue discussion. https://code.google.com/p/csipsimple/issues/detail?id=81.

[11] Discussion about proper rules of using GPS. https://github.com/SecUpwN/Android-IMSI-Catcher-Detector/issues/87.

[12] Instantupload bug in ownCloud - continuous retry but file exists on server. https://github.com/owncloud/android/issues/554.

[13] Issue in podax: Default to wifi only download. https://github.com/thasmin/Podax/issues/13.

[14] Location provider drains battery. https://github.com/rtreffer/LocalGSMLocationProvider/issues/8.

[15] Multitasking the Android way. http://android-developers.blogspot.com/2010/04/multitasking-android-way.html.

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

[17] Patch to fix aggressive widget update period configuration in anki-droid. https://github.com/nicolas-raoul/Anki-Android/commit/9193806eb0546bf8d7c040ad40272d632b4d3e99.

[18] Power saving mode optimization in AnySoftKeyboard. https://github.com/AnySoftKeyboard/AnySoftKeyboard/issues/463.

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

[20] Request: Auto sync over WiFi only. https://github.com/AndlyticsProject/andlytics/issues/678.

[21] Seadroid downloads files in loop. https://github.com/thasmin/Podax/issues/13.

[22] Wakelock detector app. https://play.google.com/store/apps/details?id=com.uzumapps.wakelockdetector.

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

[24] Why you shouldn't use a task killer on Android. http://www.howtogeek.com/127388/htg-explains-why-you-shouldnt-use-a-task-killer-on-android.

[25] XDA developers. http://www.xda-developers.com.

[26] Xposed framework. http://repo.xposed.info.

[27] Adjust. Birth, life and death of an app - A look at the Apple App Store. 2014.

[28] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A system solution for sharing i/o between mobile systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14,

pages 259–272, Bretton Woods, New Hampshire, USA, 2014. ACM.

[29] 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. ACM.

[30] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 280–293, Chicago, Illinois, USA, 2009. ACM.

[31] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, Ottawa, Ontario, Canada, 2006. ACM.

[32] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21. USENIX Association, 2010.

[33] A. Chakraborty, V. Navda, V. N. Padmanabhan, and R. Ramjee. Coordinating cellular background transfers using loadsense. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, MobiCom '13, pages 63–74, Miami, Florida, USA, 2013. ACM.

[34] R. Chandra, O. Fatemieh, P. Moinzadeh, C. A. Thekkath, and Y. Xie. End-to-end energy management of mobile devices. Technical Report MSR-TR-2013-69, July 2013.

[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. ACM.

[36] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 40–52, Paris, France, 2015. ACM.

[37] Evans Data. Global developer population and demographic study. 2014.

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

[39] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 155–168, Low Wood Bay, Lake District, UK, 2012. ACM.

[40] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck. Screen-off traffic characterization and optimization in 3g/4g networks. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 357–364, Boston, Massachusetts, USA, 2012. ACM.

[41] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 253–266, Prague, Czech Republic, 2013. ACM.

[42] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 519–530, Maui, Hawaii, USA, 2014. ACM.

[43] F. X. Lin, Z. Wang, and L. Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300, Salt Lake City, Utah, USA, 2014. ACM.

[44] 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. USENIX Association.

[45] 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. USENIX Association.

[46] M. Meeker. KPCB Internet trends. 2014.

[47] A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, and G. Challen. PhoneLab: A large programmable smartphone testbed. In *Proceedings of First International Workshop on Sensing and Big Data Mining*, SENSEMINE'13, pages 4:1–4:6, Roma, Italy, 2013. ACM.

[48] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, Roma, Italy, 2013. ACM.

[49] 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. ACM.

[50] 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. ACM.

[51] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proc. of the sixth conference on Computer systems*, EuroSys '11, pages 153–168. ACM, 2011.

[52] 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. ACM.

[53] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Periodic transfers in mobile applications: Network-wide origin, impact, and optimization. In *Proceedings of the 21st International*

Conference on World Wide Web, WWW '12, pages 51–60, Lyon, France, 2012. ACM.

[54] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 235–248, Brighton, United Kingdom, 2005. ACM.

[55] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Procrastinator: Pacing mobile apps' usage of the network. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 232–244, Bretton Woods, New Hampshire, USA, 2014. ACM.

[56] 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. ACM.

[57] M. Richtel. The youngest technorati. http://www.nytimes.com/2014/03/09/technology/the-youngest-technorati.html.

[58] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, San Francisco, CA, 2004. USENIX Association.

[59] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen. Revisiting network energy efficiency of mobile apps: Performance in the wild. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, IMC '15, pages 339–345, Tokyo, Japan, 2015. ACM.

[60] 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.

[61] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178. ACM, 2009.

[62] 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. ACM.

[63] T. W. Staff. The rise of the teenage app-developer. http://theweek.com/article/index/229474/the-rise-of-the-teenage-app-developer.

[64] J. E. Vascellaro. App developers who are too young to drive. http://online.wsj.com/articles/SB10001424052702303410404577468670147772802.

[65] 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. USENIX Association.

[66] K. Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, Inc., 1st edition, 2013.

[67] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang. Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 363–372, Tampere, Finland, 2012. ACM.

[68] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114. ACM, 2010.