Carat: Collaborative Energy Diagnosis for Mobile Devices

Adam Oliner
Anand Padmanabha Iyer
Ion Stoica
Eemil Lagerspetz
Sasu Tarkoma

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-17
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-17.html

March 8, 2013

Carat: Collaborative Energy Diagnosis for Mobile Devices

Adam J. Oliner, Anand P. Iyer, Ion Stoica
AMP Lab, UC Berkeley
{oliner, api, istoica}@eecs.berkeley.edu

Eemil Lagerspetz, Sasu Tarkoma
U of Helsinki
{eemil.lagerspetz, sasu.tarkoma}@cs.helsinki.fi

ABSTRACT

We aim to detect and diagnose *energy anomalies*, abnormally heavy battery use. This paper describes a collaborative black-box method, and an implementation called Carat, for performing such diagnosis on mobile devices. A client app sends intermittent, coarse-grained measurements to a server, which identifies correlations between higher expected energy use and client properties like the running apps, device model, and operating system. The analysis quantifies the error and confidence associated with a diagnosis, suggests actions the user could take to improve battery life, and projects the amount of improvement. Carat detected all anomalies in a controlled experiment and, during a deployment to a community of more than 340,000 devices, identified thousands of energy anomalies in the wild. On average, a Carat user's battery life increased by 10% after 10 days.

1 Introduction

Mobile computing, especially smartphones and tablets, is becoming ubiquitous. Recent work [30] acknowledged the rise of a class of mobile software misbehavior: energy bugs. These bugs add to the list of causes of poor battery life that already includes system configurations, user behavior, and power-hungry apps. Significantly increased battery drain, called an *energy anomaly*, frustrates users, creates poor press for vendors, and can render devices unusable. For such a user, the goal is to understand what is using up the battery, whether or not that is normal, and what can be done.

In this paper, we present a black box method for diagnosing energy anomalies that uses all the information available to a user app on *both* the Android and iOS platforms. In addition to being a pragmatic point in the design space, our solution naturally possesses several desirable qualities:

- Software-only. Hardware solutions are expensive, require technical skill, and void warranties.
- No kernel modifications. Hacking an OS requires skill; even "jailbreaking" may result in the user bricking their device or introducing bugs or security vulnerabilities.
- Black-box apps. The user does not have access to the source code for most of the apps they run or, usually, the ability to instrument binaries.

Extensions to our method could take advantage of platform-specific information (our implementation does so), but the aim of this paper is to evaluate how far we can take diagnosis without relying on such data. Distribution mechanisms like the app stores make it easy to get instrumentation onto off-the-shelf devices, so long as that instrumentation is a standard app offering a service that users deem valuable. An app that yields insight into poor battery life—and actionable advice for improving it—meets this requirement.

Unfortunately, a single device has limited diagnostic power because there is no *a priori* specification of normal energy use. (This is in contrast to many correctness bugs; crashing is almost always bad.) The app could measure every local signal it can access and still be left with the main questions: Is the observed energy use normal? Is it abnormal but merely a consequence of local configuration parameters or user behaviors? Would changing some aspect of the system improve battery life? If so, by how much? This limitation remains even if we modified the hardware, kernel, and apps; the information is simply not present on any single device. To answer such questions, the app would need to know about the energy use and behaviors on other devices, as well.

If, instead, we had a *community* of devices, these questions would be tractable. Measurements aggregated from multiple *clients* would allow us to collect more data more quickly, account (statistically) for individual variation in configurations and usage, say whether energy use is normal, and project the impact of certain actions. Each client occasionally records the battery level and other local data. We aggregate these measurements and compare average discharge rates under different conditions, such as which third-party apps (a common source of battery problems) are running.

If the average rate while running some app A is higher than when A is not running (but any other apps may be), we call that app an energy *hog*. A hog may be caused by a coding error (e.g., it prevents the screen from dimming) or because such energy use is intrinsic to the app's function (e.g., it frequently needs to use the GPS). If an app B is not a hog, it may still be an energy *bug* on client X if the average rate on X is higher than the average on all the other clients running B. Energy bugs may be caused by a code error that only triggers under certain conditions (which our analysis tries to discover), configurations, or user behaviors.

Our method for diagnosing energy anomalies [26] uses the community to infer a specification (expected energy use), and we call deviation from that inferred specification a hog or a bug [9]. Unlike previous work, we are looking for regularity and deviation in the use of energy and leveraging this insight to characterize the abnormal use of that resource (the battery). Deviant energy use is an anomaly, regardless of the cause (e.g., coding error or user behavior).

We take a black-box approach with process-level granularity; when we observe anomalously high energy use, we implicate one or more processes. Our method further computes diagnosis trees called *MCADs*, which enable us to advise users what actions they can take to improve battery life and to estimate the amount of improvement (accompanied by error and confidence bounds).

These measurement constraints impose fundamental limitations on our method. Without visibility into apps, we cannot disambiguate whether apps are consuming abnormal amounts of energy because of a coding error, in-app configuration, or user behavior. Apps that are often used together, such as those in app suites, may be conflated by our correlation-based approach. The lack of resource-specific instrumentation with attribution (e.g., what used the network and how much), prevents certain kinds of diagnoses. Although these restrictions may seem severe, for a method that can still be distributed via the App Store, ours is *maximally invasive*. Despite the above limitations, this paper aims to show that our data is sufficient to diagnose anomalies with enough accuracy to provide actionable recommendations that improve battery life in practice.

In this paper, we present the following:

- A collaborative method for detecting and diagnosing energy anomalies by looking for deviation from typical battery use (see Section 2) and an implementation as an app called Carat for iOS and Android (see Section 3),
- Validation of our discharge rate estimation method using power metering hardware (see Section 4), and
- Results from a 340,000-device deployment, including how we both successfully detected injected energy anomalies and diagnosed thousands in the wild (see Section 5).

We conclude with a discussion of the limitations of our approach (see Section 6), an explanation of our place among the related work and how we distinguish ourselves (see Section 7), and a summary of the conclusions (see Section 8).

2 Method

Our method builds and compares conditional probability distributions of rates of energy use to look for *energy anomalies*; e.g., the rates when an app is running on a client with one OS version (the *subject distribution*) may be significantly higher than when running on clients with another OS version (the *reference distribution*). We focus on two kinds of anomalies: hogs and bugs (see Section 2.1). In Sections 2.2–2.4, we compute the magnitude of an anomaly, corresponding to the expected improvement in battery life that an average user experiencing the anomaly would see if they became like the average user not experiencing it. We quantify the error and uncertainty of these projected improvements and decrease that uncertainty by classifying measurements according to various conditions (e.g., rates taken when WiFi was, or was not, available). We generate the classifiers for an anomaly as a diagnosis tree (see Section 2.5–2.6), which we then reduce to a minimal, complete set of actionable recommendations (*MCAD*). An MCAD translates to anomaly diagnoses, such as "With C% confidence, killing app A would increase battery life by $d_1 \pm e_1$ minutes; upgrading to OS version V would increase battery life by $d_2 \pm e_2$ minutes; disabling WiFi..." and so on.

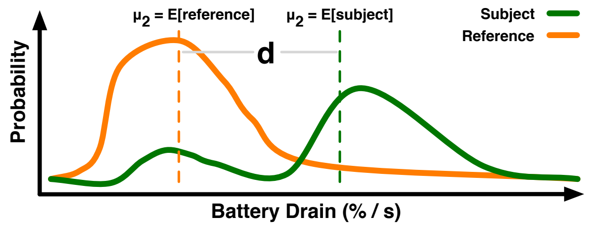


Figure 1: We compare the expected values of conditional distributions of energy drain rates to classify apps as hogs, bugs, or neither. The distance d shown is used to estimate the severity of the anomaly.

2.1 Hogs and Bugs

We define two categories of anomalies, hogs and bugs, by the types of subject and reference distributions we compare. Informally, an app is an energy hog when using that app drains the battery significantly faster, in a statistical sense defined in Section 2.4, than the average app. In contrast, an app has an energy bug when some running instances of the app (the ones in which the bug manifests) drain the battery significantly faster than other instances of the same app (the ones in which the bug does not manifest). Anomalies do not imply incorrect behavior; they may have innocuous causes. Hogs and bugs are computed as follows.

First, we build a (reference) distribution of battery discharge *rates* for devices used normally: playing games, browsing the web, making phone calls, leaving it idle, etc. Introduce an app A into the community, which some subset of clients will install and use, possibly in place of certain other apps. Build another (subject) distribution consisting only of rates observed while A is running. If the expected battery life while A is running is significantly lower than the expected lifetime without A, we call A an energy *hog*.

Intuitively, a hog lowers the community's average battery life. Note that an app may make use of energy-demanding device resources (e.g., WiFi or GPS) without being considered a hog; anomalous apps tend to overuse these resources. An app could be a hog because of a coding error that affects many clients or because an app legitimately needs to use large amounts of energy to serve its function. Regardless, a user seeking to improve their battery life would do well to not have a hog running.

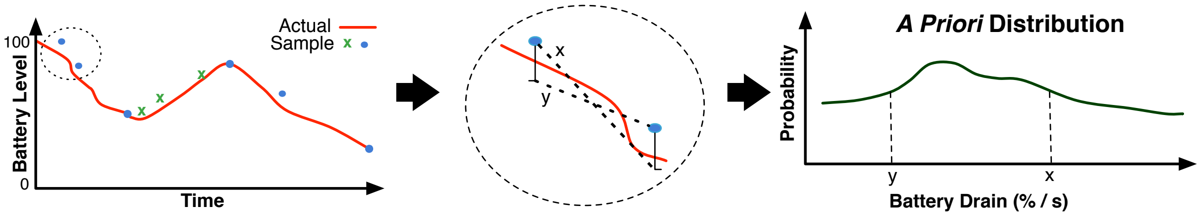


Figure 2: The process of converting battery level samples to rate distributions using the *a priori* distribution. Samples marked X are discarded because the device was charging. iOS may report a battery level up to 5% above the actual level. The slope bounds (x and y) determine the *a priori*.

An app B that is not a hog may still use much more energy on some client X. If the expected discharge rate of B running on client X (subject distribution) is significantly higher than that of B running on other clients (reference distribution), we call B an energy *bug* on client X.

An energy bug is therefore a pair: an app and a client it afflicts. An energy bug may be caused by a coding error that affects a small group of clients, a rare configuration that uses more energy ("correct" or otherwise), or unusual user behavior (which requires a community to detect). If the buggy app is getting caught in a bad state, restarting the app may return the app to normal; otherwise, the remedy is the same as for a hog. Other actions may be suggested by our diagnosis trees (Section 2.6), but the current app UI does not reflect this.

We added a caveat that a hog cannot also be a bug to distinguish anomalies that affect all or most clients (hogs) from those that affect only a select subset. Hogs are unlikely to be fixed by a restart, so the action we recommend is to kill them. This difference in appropriate response motivated the naming, and we found the distinction useful.

The subject and reference distributions are built using battery level samples from the community, as we explain in the following sections. The expected values of these distributions converge rapidly to the true expected value as the number of clients increases (see Section 5.7).

Note that even perfect knowledge of app behavior on a single client could not distinguish hogs from bugs; heavy energy use on one device could simply be a matter of configuration, user behavior, or some other bug trigger that stays static across runs. In order to say whether an app or app instance is anomalous, a community is required.

2.2 Conditional Distribution Model

As discussed in Section 2.1, to detect energy anomalies we compare two distributions of the battery drain (see Figure 1). This section explains how such a conditional distribution is modeled, and how we quantify the associated uncertainty. The input is a set of n *rates*, tuples consisting of a feature vector c and a rate probability *distribution* u, computed from some pair of samples (see Section 2.3). We model these as being randomly sampled from a true distribution U_c, with mean μ and variance σ^2, composed of measurements satisfying predicate c (e.g., iPhone 4 with WiFi access).

We first take the expected value of each u to yield a *rate* r. Consider the conditional distribution R_c of rates r satisfying c. To compute the error and confidence bounds on the expected value of R_c, we model it as n independent samples from U_c. These rates—means computed from a large number of random i.i.d. variables—are therefore approximately normally distributed as $\mathcal{N}(\mu, \sigma^2/n)$, according to the Central Limit Theorem (CLT).

This result can also be obtained by starting with the assumption that R_c is distributed as $\mathcal{N}(\mu, \sigma^2)$. Although we do not know the parameters μ and σ^2, we can estimate them using the rates $(r_1, \ldots, r_n)$. The well-known maximum likelihood estimators for these parameters—obtained by maximizing the log-likelihood function—are as follows:

$$\begin{aligned}\hat{\mu} &= \bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i \\ \hat{\sigma}^2 &= \frac{1}{n}\sum_{i=1}^{n}(r_i - \bar{r})^2.\end{aligned}$$

By the Lehmann-Scheffé theorem, $\hat{\mu}$ is the uniformly minimum variance unbiased estimator for μ: $\hat{\mu} \sim \mathcal{N}(\mu, \frac{\sigma^2}{n})$.

This agrees with the CLT method. The estimator $\hat{\sigma}^2$, however, is biased, so we apply Bessel's correction to obtain the uniformly minimum variance unbiased estimator for the sample variance:

$$s^2 = \frac{n}{n-1}\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2.$$

By our normality assumption, we can construct the t-statistic $t = (\hat{\mu} - \mu)/(s/\sqrt{n})$, which has the Student's t-distribution with $n - 1$ degrees of freedom. We can approximate the error bounds on this estimate of μ using a standard formula, where h is chosen according to the desired confidence level:

$$\mu \approx\in \left[\hat{\mu} - \frac{hs}{\sqrt{n}}, \hat{\mu} + \frac{hs}{\sqrt{n}}\right] = \hat{\mu} \pm \epsilon$$

For 95% confidence error bounds, $h = 1.96$; we use this value for all experiments in this paper. Crucially, to estimate the mean μ and to assign error and confidence bounds to that estimate, we require only the rates r, not the original distributions u.

As we gather more data, the uncertainty associated with these expected values decreases. We gauge empirically how convergence occurs in practice in Section 5.7.

2.3 Computing Rate Distributions

To compute rate distributions, our method must first convert a set of *samples* from a single client into a set of *rates*. A

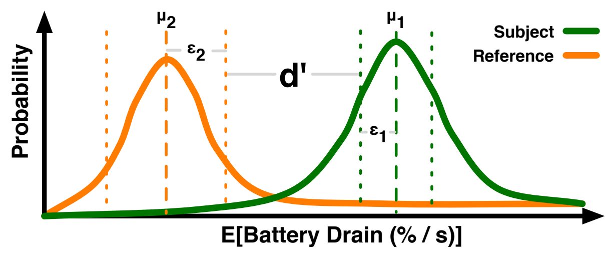


Figure 3: We compare distributions of the expected values of battery drain to identify anomalies ($d' > 0$) and quantify the error and confidence ranges for expected battery drain under different conditions.

sample is a measurement taken at a particular point in time that consists of the battery level (%) and a list of features: device model, OS version, names of running processes, battery state (e.g., unplugged), etc. Let $s_t = (b, p, q, \hat{c})$ denote a sample taken at time t, triggered by reason q (e.g., the device was unplugged), where the battery level was observed to be at fraction $0 \leq b \leq 1$ and the battery state was p (e.g., unplugged). The remaining features are denoted collectively as a set $\hat{c}$ of key-value pairs (e.g., "OSVersion=5.0" or "AppXRunning=YES").

First, we sort the samples by t and filter them (using the p values) to retain only those adjacent samples that span a period during which the device was not plugged in, restarted, or otherwise increasing in battery level: that is, only periods during which the battery was discharging. This reduces the initial set of all samples to a set of *consecutive pairs*. We compute discharge rates from these pairs.

Our method allows for imprecision in both the battery level and time measurements by converting a consecutive pair $s_{t_1} = (b_1, p_1, q_1, \hat{c}_1)$ and $s_{t_2} = (b_2, p_2, q_2, \hat{c}_2)$ not to a single rate number but to a rate distribution u. We associate this distribution with a set of features, yielding the pair $R = (u, c)$, computed from the features of the constituent pair of samples, as explained below.

If both endpoints, (b_1, t_1) and (b_2, t_2), are exact, then the rate distribution is $u = \frac{b_1 - b_2}{t_2 - t_1}$ with probability 1. Discharging yields a positive rate.

On iOS, we only get such exact measurements when the `UIDeviceBatteryLevelDidChangeNotification` is triggered. Otherwise, we estimate a probability distribution for the rate. There are a variety of techniques one might employ, depending on the nature of the uncertainty. In this paper, we address the case of iOS measurements, which present unique challenges. Specifically, the API provides battery level measurements at a granularity of 0.05. In other words, if we request the battery level at an arbitrary time during execution and get 0.95, the true level may be in the range $(0.90, 0.95]$.

The true rate, therefore, lies between $\frac{b_1' - b_2}{t_2 - t_1}$ and $\frac{b_1 - b_2'}{t_2 - t_1}$, where $b_1' = b_1 - 0.05$ and $b_2' = b_2 - 0.05$, and subject to the constraint that the rate is nonnegative. Not all values in this range are equally likely, however, so we use this range to take a "slice" of an *a priori* rate probability distribution (see Figure 2), computed using the rates that clients were able to compute exactly, as described above. There was sufficient data in this distribution to bootstrap our method. We convert the slice to a probability distribution by dividing by the slice mass and use it as the rate distribution u.

We compute c from $\hat{c}_1$ and $\hat{c}_2$ by taking the union: $c = \hat{c}_1 \cup \hat{c}_2$. Features like device model do not change between consecutive samples. We conservatively say that an app was running during the period $[t_1, t_2]$ if it was seen in either sample. It would be straightforward to use a different function if the semantics of the features demanded it.

2.4 Comparing Rate Distributions

Let c_1 be the conditions of the subject distribution (e.g., app A is running) and c_2 be the conditions of the reference distribution (e.g., app A is not running). We aim to ascertain whether c_1 corresponds to *significantly greater* energy use than c_2. For this to be answered in the affirmative, we require the following:

$$\hat{\mu_1} - \frac{hs_1}{\sqrt{n_1}} - \hat{\mu_2} - \frac{hs_2}{\sqrt{n_2}} = \hat{\mu_1} - \hat{\mu_2} - (\epsilon_1 + \epsilon_2) \quad > \quad 0.$$

Otherwise, the data does not support the assertion with the desired confidence. Graphically, this corresponds to a positive value of d' in Figure 3.

Carat suggests actions that would improve battery life along with the expected value of that improvement for an average client (starting from full charge and fully draining the battery). The improvement if the client were to change from c_1 (experiencing the anomaly) to c_2 (not experiencing it) follows directly from the distance metric $d = \hat{\mu_1} - \hat{\mu_2}$. Within our confidence bounds, however, the value of d could be as much as

$$e = h\left(\frac{s_1}{\sqrt{n_1}} + \frac{s_2}{\sqrt{n_2}}\right).$$

This is symmetric about the expectation. The estimated improvement is therefore $d \pm e$.

2.5 Splitting Distributions

In order to more confidently diagnose anomalies, we build a tree that separates conditional distributions by features that significantly affect energy use. Let each conditional distribution be a node in this tree, uniquely identified by its condition c. Starting with some distribution c (e.g., app A is running), iterate through each feature $f \notin c$ and attempt a split by creating new child nodes $c \wedge f$ and $c \wedge \neg f$. For instance, if f is whether the client is running a Galaxy S II, then one child would get the rates from node c taken from Galaxy S IIs and the other would get all other rates satisfying c.

Splitting has two competing effects on the error bounds. First, it reduces n, thereby increasing the error (increasing uncertainty). Second, if feature f divides rates from distributions having significantly different means, then it will likely reduce the sample variance of at least one child and thereby decrease the error (decreasing uncertainty).

A split is performed if the child nodes c_1 and c_2 yield a positive gap, $d' > 0$, as in Figure 3. Splitting generates

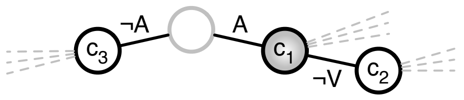


Figure 4: The minimal complete actionable diagnosis (MCAD) for the example anomaly c_1 described in Section 2.6, consisting of c_2 and c_3. The dashed lines indicate nodes and subtrees that, while produced via splits when the tree was constructed, did not meet the criteria for an MCAD.

two leaves, children of c, with edges f and $\neg f$. Otherwise, we make no changes to the tree and proceed to test the next feature. When no more features remain, we can recursively repeat the process on any new leaves.

2.6 Diagnosis

This section describes how to generate a diagnosis for an anomaly, which involves building a tree structure similar to a classification or decision tree [23, 38], and conclude with an example. Consider a node c_1 corresponding to a subject distribution for an anomaly (see Section 2.1). A *diagnosis* is a set of nodes with significantly lower energy use than c_1. Intuitively, a node in this diagnosis is some condition under which the anomaly does not occur. The diagnosis is *complete* if it includes all such nodes.

Let node c_2 be said to be *reachable* from node c_1 if, in the problem domain, it is possible to initially be in a state satisfying c_1 and, by performing some actions, then satisfy c_2. We define an *actionable* diagnosis to be one consisting only of reachable nodes.

A diagnosis is *minimal* if every subtree entirely contained in a complete diagnosis is replaced by its root. The minimal complete actionable diagnosis (MCAD) is unique, but note that it may include paths from c_1 to multiple different states.

For example, consider the node for running app A, $c_1 = A$, with significantly more energy use compared with $\neg A$; it is a hog. Say, for simplicity, that there are only two other features of the device—model M and OS version V—and only one other possible OS version. Every node in the subtree rooted at $\neg A$ has significantly lower energy use than c_1, as does every node with $\neg M$ or with $\neg V$. In our domain, a user cannot change their device model, so all nodes with $\neg M$ are excluded from the actionable diagnosis despite showing less energy use. To make the diagnosis minimal, replace with their respective roots the nodes in the subtrees rooted at $A \wedge \neg V$ and $\neg A$. Thus, the MCAD (illustrated in Figure 4) is exactly these two nodes (c_2 and c_3); the interpretation is that the client can improve their battery life either by changing OS versions or killing the hog.

These trees helped diagnose problems in the wild, such as the Kindle bug in Section 5.6 where WhisperSync was using far more energy when syncing over GSM. Our analysis discovered the bug was correlated with the iPhone 4 and only occurred on iPads when they did not have WiFi.

3 Implementation

The Carat architecture consists of a mobile app (see Section 3.1), central server (see Section 3.2), and analysis running in the cloud (see Section 3.3). Figure 5 shows an overview.

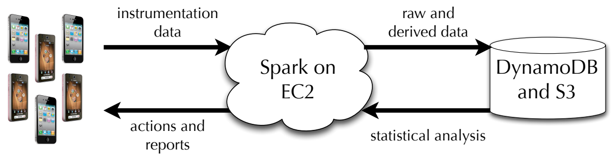


Figure 5: The Carat architecture, consisting of the crowd-based front end, the central server with the analysis running in the cloud, and the stored samples and results.

3.1 Carat App

We implemented Carat as an app on both the iOS and Android platforms. It is available as a free download on Apple's App Store, Google's Play Store, and as source code on GitHub, all of which are linked from the project homepage[1]. The clients are lightweight; e.g., the iOS app is ∼6000 lines of Objective-C, excluding third-party libraries like Flurry (for collecting usage statistics), ShareKit (for enabling sharing over social networks), Thrift (for handling messaging protocols), CorePlot (for plotting), and several others. This number also excludes auto-generated code related to the UI.

Carat runs as a user-level app on stock devices. This places platform-specific restrictions on what information is accessible and when our app is allowed CPU time to measure it. Our implementation records the following information using the public APIs:

- battery level fraction,
- battery state (e.g., plugged in or unplugged),
- names of running processes (each non-OS process roughly equates to a single user app),
- state of memory (e.g., number of active pages),
- OS and version,
- device model, and
- a unique, anonymous, Carat-specific client ID.

This information resides in persistent storage until the app is brought to the foreground, at which point it communicates with the Carat server over TCP. Our communication model is client-initiated (since they are situated behind NATs) and utilizes Apache Thrift to define the service interface.

The app intermittently transfers stored samples to the server over 3G or WiFi. Since we optimized Carat with respect to energy use, the client invokes a data transmission to the server only when it is running in the foreground and when the user is interacting with the UI. At this time, the app also requests results from the server to update the UI.

To comply with legal restrictions and to alleviate user concerns, our implementation neither records nor transmits personally-identifying information. What it does record is viewable within the app (see Section 3.1.1), so the user knows exactly what Carat is measuring. Furthermore, our EULA (required by the App Store and also available on the project webpage[1]) includes an additional clause making it clear exactly what our app will do. Furthermore, the app is open source under a BSD license and is available on GitHub[1].

Although jailbroken iOS devices would have allowed us to collect more data (e.g., app versions), it also would have restricted the size of our user-base, biased our data toward a certain class of users, and prevented us from distributing Carat on the App Store. We opted for less data from more users, and our results demonstrate that energy anomalies can be detected without intrusive instrumentation.

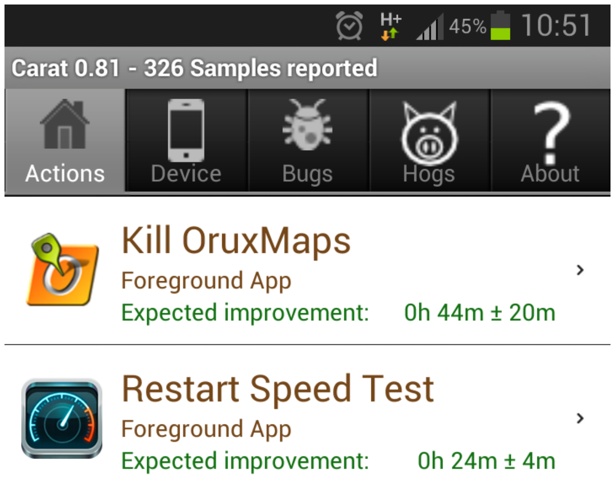
On Android, Carat samples when the `ACTION_BATTERY_CHANGED` intent fires, at 1% battery level granularity. As we discuss for the remainder of this section, not only is Carat more restricted on iOS than Android with respect to what it can measure, but also when. Carat does not fall into the class of apps that are allowed to run as proper background tasks, which are given intermittent CPU time to perform tasks such as buffering audio, maintaining VoIP server connections, or continuously tracking the GPS coordinates of the device using location services. This means that, in order to take samples while Carat is suspended, our app subscribes to several notifications. When one of these notifications is triggered, iOS allows Carat a small amount of time to take measurements and save these to persistent storage; there is not enough time to communicate with the server.

Carat subscribes to battery-related events (`UIDeviceBatteryLevelDidChangeNotification` and `UIDeviceBatteryStateDidChangeNotification`) and significant location changes (`startMonitoringSignificantLocationChanges`). The location change feature is especially valuable for us. It not only uses far less energy than using the full-fledged location service, but it means that the OS will automatically relaunch Carat if it is terminated while the service is active. (In our deployment, while Carat was in the background, roughly half of samples were triggered by location services and a third were triggered by the battery level event.)

3.1.1 User Interface

When the Carat app is launched, it sends locally stored samples to the server. When Carat is in the foreground, the temporal resolution of sampling increases several-fold. These observations—that increased user engagement leads directly to data being recorded more often and reported sooner—motivated us to spend time honing the user interface, which we now present.

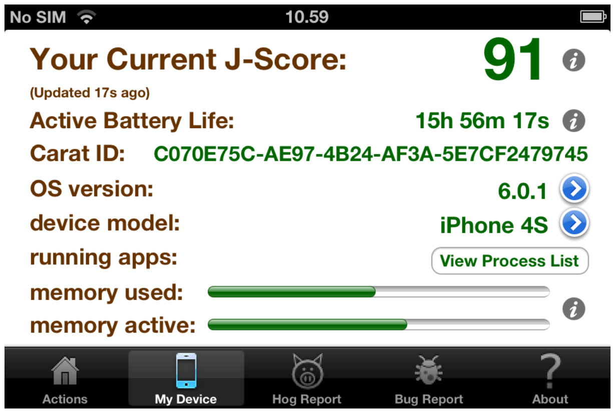
The main screen of Carat is the Actions list, shown in Figure 6, which presents actions the user can take to improve battery life, based on what Carat has learned about their device (e.g., what apps they run), sorted by the expected improvement if that action is taken. For example, the figure shows an action "Kill OruxMaps" that would result in an expected increase of 44m. This means our analysis observed that a typical device running this game will run a full battery down to zero almost 44 minutes sooner than a typical device running typical apps but not OruxMaps. Carat will suggest restarting bugs, admitting the possibility that the instance is caught in a bad state; if restarting does not help, it may be a configuration problem or specific to user behavior. Finally,


45% 10:51
Carat 0.81 - 326 Samples reported

Kill OruxMaps
Foreground App
Expected improvement: 0h 44m ± 20m

Restart Speed Test
Foreground App
Expected improvement: 0h 24m ± 4m

Figure 6: The top of the main screen of Carat on Android, showing recommended actions and projected battery life improvements.


No SIM 10.59

Your Current J-Score:	91
(Updated 17s ago)	
Active Battery Life:	15h 56m 17s
Carat ID:	C070E75C-AE97-4B24-AF3A-5E7CF2479745
OS version:	6.0.1
device model:	iPhone 4S
running apps:	View Process List
memory used:	
memory active:	

Figure 7: The Device tab on the iOS client. The J-Score indicates the percent of the community with worse battery life than this device.

our current implementation will suggest upgrading the operating system if it observes that a newer version is correlated, across the community, with better battery life. The current UI does not reflect all information present in the diagnosis trees; that is planned for a future release.

The Device tab displays information about the client's device, including most of the information that is being recorded and transmitted to our server: the process list, the device model and OS, the state of memory, etc. This tab also prominently displays a number called a J-Score, which is the percentile into which the client's battery life falls within the community; a J-Score of 65 means a better active battery life than 65% of devices. Active battery life is computed based on Carat sampling and omits idle periods. This client's average battery drain when using the device would fully deplete the battery in about 16 hours.

We created the J-Score (see Figure 7) to increase user interest and sharing, hoping that it would introduce an element of social competitiveness to energy efficiency. It appears, anecdotally, to have worked. For instance, upon observing that her score had dropped precipitously due to an influx of new users, one user remarked (tongue-in-cheek) that she was "no longer confident in our analysis results." She continues to check her score regularly, incidentally sending us samples each time.

The Actions list only suggests killing or restarting an app that is currently active (i.e., in the process list). The Hogs tab shows the top hogs ever reported to have run on the device. The same is true for bugs under the Bugs tab. Clicking on one of the hogs or bugs brings up a detail page where the user can explore the data further.

3.2 Carat Server

The Carat server collects samples from instances of the Carat app running on clients' mobile devices and stores them for use by the backend analysis (see Section 3.3), and it serves actions and other analysis results to clients.

The server is a <1300 line Java application (excluding code auto-generated by Thrift) that listens on TCP port 8080 for incoming client connections. We host the server on Amazon EC2 because it provides a mechanism to scale the server by spawning new instances and to run a load-balancer to distribute incoming connections.

Received samples undergo lightweight processing to remove junk or malformed data and are then sent to persistent storage. This preprocessing removes OS daemons from the list of processes. We manually maintain a blacklist of such daemons, as it does not appear that the iOS API provides enough information to determine this automatically.

Our storage layer uses Amazon's DynamoDB, which lets us retrieve key-based information (e.g., chronological samples from each client) with little performance overhead, along with the freedom to change attributes in stored entries without changing the structure of the table (e.g., adding a new sample field). We also retain a backup of our data in S3.

3.3 Backend Analysis

The Carat analysis consists of approximately 5000 lines of Scala, written in the Spark framework [43]. The production version runs in a 20-node cluster composed of high-memory Amazon EC2 instances. This section provides an overview of Spark, the challenges related to parallelizing Carat, and our solutions.

After converting samples to rates, the analysis proceeds in two main stages: identifying hogs and bugs and then generating MCAD trees (see Section 2). The first stage is summarized in Algorithm 3.1

3.3.1 Spark Overview

Spark is a cluster computing framework designed for iterative and interactive jobs. Existing data-flow based frameworks such as Hadoop or Dryad depend on intermediate data being written and read from disk, incurring a huge performance hit for iterative jobs. In contrast, Spark provides an efficient environment for multi-stage jobs by reusing the same worker nodes across iterations. In addition, it provides a robust programming model for interactive queries where it is desirable to load data into memory and query it repeatedly (with different filters).

Parallel programming in Spark is provided in the form of Resilient Distributed Datasets (RDDs), which are read-only collections of objects partitioned across a set of machines that can be rebuilt if a partition is lost, and a set of parallel operations on the RDDs (e.g., `foreach`, `reduce`, and `collect`). These features, along with fault tolerance and its memory management model, made Spark a good fit for implementing Carat's analysis.

Algorithm 3.1: ANALYZERATES($allRates, aDist$)

```
comment: Hog detection
for each $app \in allApps$
  do
    $filt \leftarrow$ ALLRATES.FILTER(app in _.allApps)
    $filtNeq \leftarrow$ ALLRATES.FILTER(app not in _.allApps)
    $pDist \leftarrow$ GETDIST($filt, filtNeq, aDist$)
    if $pDist.evDistance > 0$ && ISSIGNIFICANT($pDist$)
      then comment: store hog and distributions
comment: Bug detection
for each $id \in allIds$
  do
    $fid \leftarrow$ ALLRATES.FILTER($\_.id = id$)
    $notFid \leftarrow$ ALLRATES.FILTER($\_.id$!=$id$)
    comment: Consider apps reported by id, omit hogs
    $fidNonHogs \leftarrow$ FID.MAP($\_.allApps$) $\setminus Hogs$
    for each $app \in fidNonHogs$
      do
        $appFid \leftarrow$ FID.FILTER(app in _.allApps)
        $appNotFid \leftarrow$ NOTFID.FILTER(app in _.allApps)
        $pDist \leftarrow$ GETDIST($appFid, appNotFid, aDist$)
        if $pDist.evDistance > 0$ && ISSIGNIFICANT($pDist$)
          then comment: store bug and distributions
    $scoreDist \leftarrow$ GETDIST($fid, notFid, aDist$)
    comment: Save scoreDist for J-Score calculation
comment: Write J-Scores based on the processed distributions
```

3.3.2 Parallelizing Samples to Rate Conversion

In Section 2.3, we discussed how Carat converts consecutive samples from users into rates. This computation involves a dependency between samples that complicates the parallelization process.

To remove this inter-sample dependency, we create RDDs of consecutive sample pairs. This new RDD is free of dependencies, so the Spark runtime can independently assign data and conversion tasks to workers. This is done by applying a `map` operation to every item in the RDD. The result of this operation is another RDD consisting of rates. We add metadata for backtracking.

3.3.3 Parallelizing Distribution Building

The bulk of Carat's analysis is the process of building and comparing rate distributions. To reduce memory requirements, we load the rates into an RDD (see Section 3.3.2). Spark automatically distributes the RDD to all compute nodes, thus optimizing memory usage per node. To achieve optimal parallelism, the strategy must compute distributions on features in parallel. That is, when building distributions on feature c, the technique must compute distributions for all values of feature c. We devise such a strategy using Spark's RDD operations as follows.

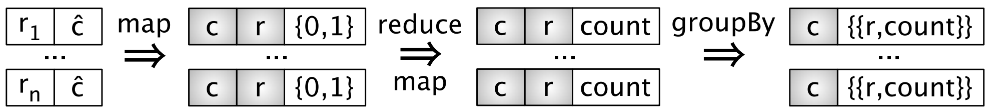


Figure 8: The parallelization process starts with rates as an RDD. Each rate r has a set of features $\hat{c} = (c_1 \dots c_n)$. To compute rate distributions on feature c (e.g., each app), we first map the RDD to a structure with (c, r) as the key (shaded) and, as the value, 1 if the feature occurs and 0 otherwise. A reduce operation yields the rate frequencies for features. We map again, now with c as the key, and (r, count) as the value. Grouping by key then gives the frequency of every R for every F. With slight modifications to the mapping and grouping fields, we use this parallelization strategy for hogs, bugs, J-Scores, etc.

We begin with items in the rate RDD, composed of rates r and their associated features $(c_1, ..., c_n)$, split among worker nodes. We compute distributions of rates conditioned on c and compare them with distributions satisfying $\neg c$. (We compute the distribution for $\neg c$ by subtracting the distribution for c from the full distribution.)

The first step maps items to the format $((c, r), \{0, 1\})$, keyed on c and r and with a value of 0 or 1, indicating the presence of the rate. This is computed from the apriori (see Section 2.2). A `reduce` operation computes the frequency of each such (c, r) pair. We remap the reduced RDD and make c the key and (r, count) the value. When we apply a `groupBy` operation on the key, we obtain the frequency of every rate for every value of c, or a sequence of $(c, (r, \text{count}))$ (see Figure 8).

We now have two RDDs, one which has the frequency of rates satisfying c and its complement. The RDDs are joined using a `groupWith` operation. A final `map` operation passes them through our distribution building and comparison module in a parallel fashion, thus obtaining the expected improvements and the correlations. These results are stored in DynamoDB where they are retrieved on-demand by the client.

The same parallelization strategy is applied to compute hogs (features are apps), bugs (features are (UserID, App) pairs), J-scores (features are UserIDs). We observe that most other feature-grouping required in Carat's analysis can be reduced to this parallel model.

4 Ground Truth and Overhead

In order for Carat to accurately account for when energy is being used, it must convert intermittent (low precision) battery level samples into energy drain rates in a way that is faithful to the ground truth. Furthermore, the practicality of our method relies on sampling that is sufficiently low-overhead that it does not have a significant impact on the energy use, itself. In this section, we attach mobile devices to power metering hardware: an iPhone 4S to a Monsoon Power Monitor[2] (see Figure 9) and a Galaxy Tab 2 10.1 to Leyden Energy's[3] battery-testing equipment. Our results confirm that Carat generates accurate energy distributions while consuming few resources (i.e., almost no battery).

To test the fidelity and cost of our sampling, we ran the devices through scripts of varied activities. The scripts are not intended to be a representative workload, but to repeatably exercise the device features and drain the battery at different rates. It includes such behaviors as downloading and running an app, browsing the web, playing a game, and idle periods.

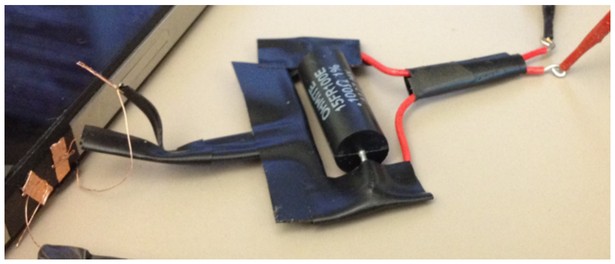


Figure 9: Close-up of the wiring rig that connects our iPhone 4S test phone with the Monsoon Power Monitor.

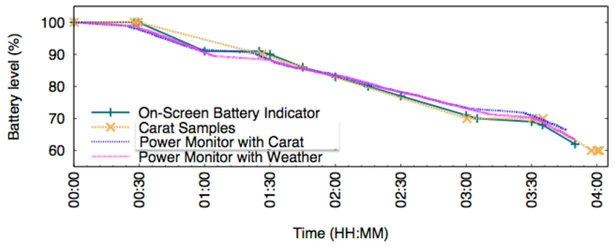


Figure 10: The battery levels during our iOS power metering experiments, either taken directly from the on-screen battery indicator, the Carat samples, or computed from the meter's readings.

The WiFi was turned on for some periods and off for others.

On each device, we ran through the script under three different arrangements: (1) hooked up to the power meter with and (2) without Carat running and (3) not hooked up to the power meter with Carat running. We compare the data from (1) and (2) to quantify the overhead of running Carat; we compare the data from (1) and (3) to ensure the meter was not influencing Carat's measurements and to assess the fidelity of our sampling and rate estimation. For the runs performed without Carat, where our app appears in the script, we substituted the standard Weather app.

The battery levels reported by the OS, both through the API (Carat samples) and the on-screen indicator, track the actual use of power by the device. Figure 10 shows the iOS data. Between 00:30 and 1:30, Carat took no samples and conflated a higher-rate period with a lower-rate period. Higher frequency sampling would have avoided this error.

The expected energy discharge rates computed from the Carat samples approximate the values computed using power metering hardware. During the 9-hour iOS experiment, Carat took 9 samples at 5% granularity; the power meter took 13,549 samples at effectively 0.0001% resolution. Carat overestimates the average discharge rate by only 0.00088%/sec (see Figure 11). On the Galaxy Tab, where Carat took twice as many samples as on iOS (19), the error is an order of magnitude less (0.00015%/sec). This accuracy is possible thanks to the *a priori* distribution, which uses knowledge

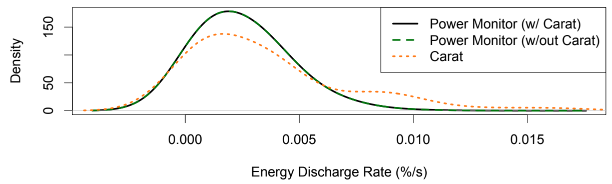


Figure 11: The energy rate distributions from our iOS power metering experiments, smoothed with a Gaussian kernel estimator for visibility. Using the *a priori*, Carat is able to faithfully estimate the distribution with sparse sampling, overestimating the mean energy drain rate by only 0.00088% from 9 samples.

of community behavior to refine noisy and incomplete measurements; imprecision in per-client measurements is further mitigated by the statistical backend analysis.

Carat imposes negligible energy overhead. Our power metering hardware indicates that running through our iOS script with Carat running used *less* energy (53.691 mAh or ∼3.5% of the battery less) than executing that same script with the Weather app running in its place (i.e., 54 minutes less battery life running Weather instead of Carat). We also ran the script without substituting another app but found battery life with Carat running was slightly higher than without; Carat's energy use is less than the experimental imprecision. Similar results held on Android. We can afford to perform such sparse, low-overhead sampling on individual clients because we aggregate such data from many clients.

5 Deployment Results

Carat became available as a free download on Apple's App Store and on Google's Play Store in mid-June of 2012. Days later, it was featured on the popular TechCrunch blog^4; the story was immediately picked up by dozens of other news sources. Within 24 hours of the article's publication, we went from a few hundred users to more than 100,000. This doubled in the subsequent 24 hours. More than 340,000 unique devices have run Carat.

5.1 Data

Of the 340,000+ clients that ran Carat, at the time of writing, 270,063 had reported enough data for our analysis to produce results for them. Our users were 58% iOS and the rest Android. Tables 1 and 2 show a breakdown of the most common device models and operating systems. The community recorded roughly 8.6 million rates, launching the app 3.5 million times (a median of 1.9 sessions per day).

The community ran 119,652 different apps, with a disproportionate number (77%) coming from Android users. Of these apps, 11,256 (9.4%) were classified as hogs, of which 86% were Android apps. Carat detected energy bugs in thousands of apps; of the 9,136,237 total possible bugs (user-app instance pairs), 5.3% were classified as such. These data suggest that Android apps that deplete the battery generally do so uniformly across clients, while iOS apps have great variance in energy use from one device to another.

Clients reported samples at a wide variety of rates, clustering into casual users recording a few samples daily and heavier users sampling sometimes a hundred times as often.

Device Model	Number	% Total	% Platform
iOS			
iPhone 4S	62,831	23.3	39.8
iPhone 4	45,713	16.9	28.9
iPhone 3GS	10,318	3.82	6.53
iPad 2 (WiFi)	6940	2.57	4.39
Verizon iPhone 4	5662	2.10	3.58
Other	26,479	51.31	16.8
Android			
GT-I9100	12,046	4.46	10.7
unknown	10,812	4.00	9.64
Galaxy Nexus	7305	2.70	6.52
GT-N7000	4152	1.54	3.70
GT-I9300	3866	1.43	3.45
Other	73,939	85.87	65.98

Table 1: The most common device models in our deployment, showing the percent of users from whom we had sufficient data.

OS Version	Number	% Total	% Platform
iOS			
5.1.1	135,880	50.3	86.0
5.0.1	8882	3.29	5.62
5.1	6172	2.29	3.91
6.0	4374	1.62	2.77
Other	2635	42.50	1.70
Android			
4.0.4	21,650	8.02	19.3
4.0.3	19,121	7.08	17.1
2.3.6	14,355	5.32	12.8
2.3.4	13,164	4.87	11.7
Other	43,830	74.71	39.10

Table 2: The most common operating system versions in our deployment, showing the percent of users from whom we had sufficient data.

The average number of samples per day was nearly the same on both platforms (36.8 samples per user per day on iOS and 37.7 on Android), but the variance of this rate on Android was 32% higher than on iOS. This is, in part, because some Motorola devices only triggered the battery level intent at 10% levels while most other Android devices triggered every 1%; iOS devices triggered consistently at 5% increments.

5.2 User Behavior

The frequency and duration of user engagement matters. The more often users launch Carat, the fresher our data will be (that is when it is sent to our server). On both iOS and Android, the longer users keep Carat in the foreground, the more samples it can record. The session length data (see Table 3) and click-path data show that many stay in the app to explore the reports or check their J-Score. The majority of sessions last more than 30 seconds. After a month, we retain roughly 25% of our users. The median user opens Carat 1.9 times per day and 3.0 times per week.

5.3 Performance and Scaling

The success of our approach depends on an active community and generates better results as that community grows, so the implementation must be scalable.

Our frontend experienced linear traffic scaling with the size of our deployment, at a rate far below 1 byte per second per client. Sample reporting is presumed to be unreliable; a client with no disk space or network access is al-

Session Length	Sessions	% of Sessions
0–3 secs	129,860	3.81
3–10 secs	478,930	14.05
10–30 secs	1,152,333	33.82
30–60 secs	776,364	22.78
1–3 mins	618,721	18.16
3–10 mins	90,809	2.66
10+ mins	160,578	4.71

Table 3: The length of Carat sessions. The app only reports data when it is opened and can sample more aggressively in the foreground. So, incentivizing the user to open the app and explore results from within the UI helps us collect more data.

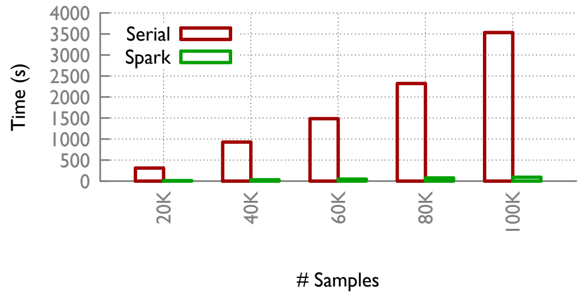


Figure 12: The Carat analysis scales almost linearly when parallelized, while a serial implementation shows exponential complexity.

lowed to throw away samples and an overloaded server may drop packets. Five medium Amazon EC2 instances behind an Elastic Load Balancer (ELB) has been handling our userbase of 340,000 devices.

Our current implementation of the analysis backend (see Section 3.3) uses the Spark cluster computing framework. The computation is massively parallel, as every distribution and comparison can be computed independently. Figure 12 compares the runtime for an optimized serial implementation of the analysis algorithm compared to a parallel implementation in Spark for increasing number of samples. The results underline the need for parallelization. As our userbase grew, we made numerous optimizations. The analysis program now computes all reports for all our users (24 million samples) from scratch in approximately 45 minutes.

5.4 Injected Anomalies

We added energy anomalies to an existing app—initially with no apparent misbehavior—to confirm that Carat is able to detect the new bugs. For these controlled experiments, we used a private deployment of 75 devices. (In subsequent sections, we detect anomalies in the wild.)

We chose the Wikipedia Mobile app made by Wikimedia Foundation because it is an open-source app used by many of our clients but was not reported as an anomaly. We added several behaviors that consume large amounts of energy when activated, with each one repeatedly using a different resource: radio, CPU, and GPS.

We installed this buggy instance on one of our test devices, an iPhone 3GS. Wikipedia Mobile was already in use by a handful of clients at this point, so a baseline distribution had been established and Carat did not consider the app to be anomalous. We ran the app for one day for each injected bug, activating the app a handful of times during the

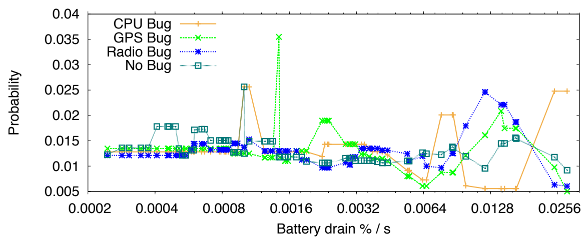


Figure 13: The reference (anomaly-free) and anomalous rate distributions for the modified Wikipedia Mobile, using only the a priori from the private deployment. Carat successfully detects all of the injected bugs.

day but only leaving it open for a couple of minutes (casual use). At the end of the third day, we ran the analysis with the real, non-buggy data as the reference distribution and once each with the data from exactly one of the buggy days as the subject distribution. Thus, we could declare success if the analysis reported three bugs, one for each injected behavior.

Indeed, after performing the injection, Carat correctly detected each of the three bugs (no false negatives). Figure 13 shows the reference distribution and each of the three subject distributions for the iPhone 3GS running our buggy Wikipedia build. The expected improvement reported for fixing each bug (i.e., returning the app to typical Wikipedia Mobile behavior) was 27m 26s for the CPU bug, 9m 22s for the GPS bug, and 55m 28s for the Radio bug, which agreed with what the experimenter observed on the device.

5.5 Hogs

Of the 119,652 apps seen during our deployment, 11,256 (9.4%) were categorized as hogs. (Before checking for statistical significance, there were 15,038 (12.6%).) Recall that an app is a hog if the community-wide average discharge rate while running the app is significantly greater than the average rate while not running it (see Section 2.1) and that we can compute the expected improvement in battery life by killing a hog (see Section 2.4). Hogs may be caused by an oft-triggered code bug or may be simply intrinsic to the app. Users concerned about battery life are advised by the Action list to kill hogs; the user is not concerned about the intention, or lack thereof, behind the energy use.

We now describe a couple of hogs and cite corroborating evidence that the app does, indeed, consume an anomalously large amount of energy. For every hog reported by Carat that we checked (several dozen, admittedly a small fraction of the total), we could corroborate the classification with one or more of user complaints, news coverage, personal experience, or experimental results in the literature (e.g., [31, 32]).

Pandora Radio: Carat classifies Pandora Radio, which 7561 iOS users ran, as a hog and says killing it will increase an client's average battery life by 50m 43s. This is corroborated by user reports, one of which claimed Pandora drained the battery to 30% in a few hours even with the screen off[5]. To improve battery life while using Pandora, the MCAD suggests using WiFi for connectivity (an additional 25–35m). If a Pandora user is on the move, the best approach is to turn off WiFi and use the mobile network for

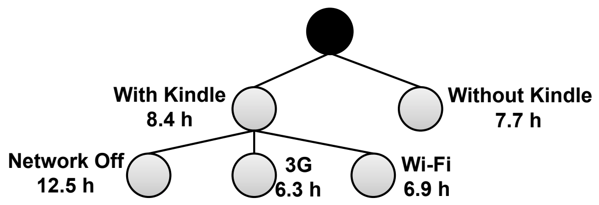


Figure 14: MCAD for the Kindle app on iOS, showing the expected battery life when using exclusively this app under various conditions. The diagnosis points to network connectivity as the primary determinant of energy use.

connectivity (+25–33m).

Skype: 43,716 iOS clients were running the Skype VoIP app, which was also reported as a hog. This is also confirmed by the forums; one user even used the term 'power hog' to describe Skype[6]. Skype's energy use is driven by network connectivity; when no network connection is available, expected battery life is about 6.5h above average.

Go launcher exe new theme...: (*sic*) Is a hog on the Android platform that costs most users between 2h 1m and 2h 53m of battery life. Experiences with Go Launcher and its variants, which change the UI of the device, vary among users[7], but generally "fancier" themes and widgets cause higher battery drain[8].

Live wallpapers: Carat identifies several Android Live Wallpapers as energy hogs. Two that rank among the top 10 most severe hogs on the Android platform are Beach at Night[9] and Heart and Love[10]. They cost most users 2h 33m–2h 49m and 2h 37m–2h 51m battery life, respectively. Both are ad-supported; the detrimental effect of adware on battery life is known [31]. Both live[11] wallpapers[12] and adware[13] have been blamed for abnormally fast battery drain.

5.6 Bugs

Recall that a bug is an app that is not a hog (it usually consumes below-average energy) but consumes far more energy on some clients than others (see Section 2.1). Although the current Carat client-side UI only suggests restarting a bug (in case it is simply caught in a bad state), the MCAD diagnosis computed on the backend enables more specific recommendations, such as disabling WiFi or turning on GPS; we plan to add this in later versions of the app. Note that, without a community of clients, distinguishing bugs from hogs would be impossible and identifying the triggers would be difficult.

The maximum number of bugs that Carat could report is the sum over clients of the number of non-hog apps they ran, which was 9.1 million in our dataset. Our method reported 483,354 buggy app instances (5.3%); we describe some examples below. As with hogs, we were able to corroborate investigated bugs using a variety of sources; in some cases, however, we were not able to identify the trigger of the bug and only know that it was causing problems for some users.

Kindle: This electronic book app was reported as a bug for 510 out of 13,226 iOS clients (3.9%). Figure 14 shows a diagnosis tree for Kindle, in which 3G connectivity appears especially detrimental. The support forums blame the problem on WhisperSync[14], which synchronizes notes, bookmarks, previous location, and Popular Highlights. When syncing over GSM, in particular, the device uses much more energy than syncing over WiFi. Our data support this hypothesis, which had previously been only anecdotal.

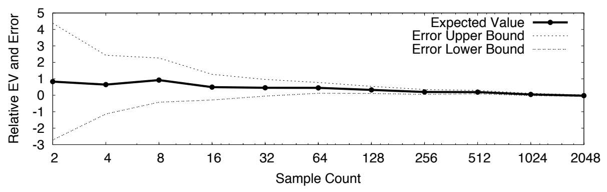


Figure 15: As the number of samples increases, the relative error in our estimate of the expected discharge rate shrinks rapidly. Above is the average expected value for several of the largest anomalies seen in our deployment and the 95% confidence error envelope.

Facebook: This mobile app was a bug for 8909 out of 79,609 iOS clients (11.2%). Higher energy use was not correlated with a particular device model or OS version (the highest correlation was 0.057). We believe this high variance in energy use may be attributable to the variety of ways that users interact with the app (that is, workload).

Facebook Messenger: Was anomalous on 792 of 7350 Android clients (10.8%). The MCAD indicates that upgrading the OS improves battery life (71–83m), and that WiFi is more energy efficient than other connectivity options. Using the app while stationary gives a 63–97m boost to battery life. (Note that Carat does not advise users to stand still.)

YouTube: Was a bug on 3118 of 37475 iOS clients (8.3%). The MCAD shows that while moving, users of mobile Internet have a battery life advantage over WiFi users (25–34m). When compared to immobile WiFi users, mobile network users still have a 20–28m advantage. This is contrary to many apps, where WiFi is less energy-consuming.

Twitter: Was reported as a bug on 2744 of 18651 Android clients (14.9%). The MCAD for Twitter indicates that the most critical cause of battery drain is an old OS version. Users of Ice Cream Sandwich (4.0.4) got 94m to 100m more battery life than other Android Twitter users. Use of WiFi with 4.0.4 yielded another 85m to 105m; this was not observed on other OS versions.

SwiftKey: A popular keyboard application for Android, SwiftKey is one of the top 15 bugs (by severity), affecting 2402 Carat users. The developer website indicates that the latest release of the app exhibits high energy drain, especially in newer versions of Android OS[15].

5.7 Result Confidence

As the number of clients and samples increases, so does the accuracy of our results. In particular, Carat's estimate of the expected value—the crucial number used to identify anomalies and compute expected benefits—tends to converge to the true value. Figure 15 shows the shrinking relative error envelope of this estimate for some of the anomalies Carat detected in the wild.

There is no guarantee of convergence in practice because the true rate distribution may be neither stationary nor identically distributed. Indeed, this paper has discussed at length

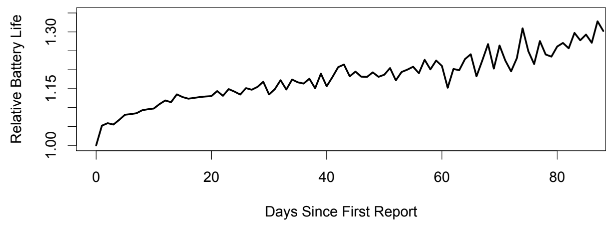


Figure 16: Average relative battery life of Carat users following the generation of their first report (hog and bug lists), using the battery life of the first day as the baseline. A typical user sees an 8.3% increase after a week, surpassing 20% after two months (as the reports fill out with more apps).

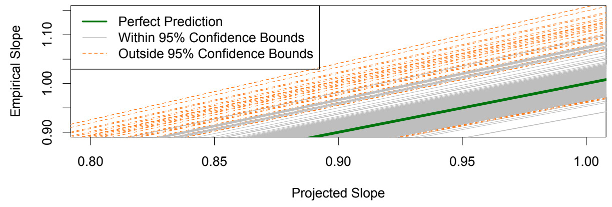


Figure 17: The projected battery life slope b_a (x-axis) and the empirical improvement slope b'_a (y-axis), as fit using a least squares regression on data from a sample of users who ran the top significant hogs. Most users (95.4%) experienced improvements within our error bounds, while the rest (orange) tended to see more improvement than Carat predicted.

one situation where a rate distribution may not be identically distributed across clients: the presence of an energy bug. As long as a bug affects a constant fraction of the population, however, this convergence happens almost surely.

5.8 Validating Recommendations

In order to verify that the recommendations provided by Carat improve battery life, we consider two metrics.

The first metric is whether battery life tends to increase over time for users of the app. This is a coarse measure of whether using Carat correlates with reduced energy use. The metric is coarse because it includes several confounding factors: some of these users may not have followed Carat's recommendations, the population is biased toward users who originally had battery problems (and thus installed Carat), and users may have also employed alternative means to decrease energy use. Figure 16 shows average relative battery life over time for Carat users. After 2 weeks, the average user sees an 11.4% improvement in battery life; long-term users (90+ days) improve by more than 30%.

The second is how closely the Carat Actions—and their projected benefits—match the observed benefits. Let $x_{u,a}$ be the fraction of the time that user u reports running app a, within some window of time. The estimated battery life improvement b_a (in seconds) that Carat quotes assumes a transition from $x_{u,a} = 1 \rightsquigarrow 0$. We assume that the achieved benefit is linear in Δx, so moving from $x_{u,a} = 1 \rightsquigarrow 0.5$ yields an improvement of $0.5 b_a$ seconds; transitioning from $x_{u,a} = 0.5 \rightsquigarrow 0.3$ yields an improvement of $0.2 b_a$ seconds. (Other actions that Carat suggests, such as upgrading the operating system, cannot be done fractionally.)

The predicted benefit b_a is a coefficient; we can evaluate the accuracy of this coefficient by comparing the predicted improvement curve $y = b_a x$ with the empirical curve, a least-squares best-fit line with slope b'_a. Figure 17 compares b_a (projected) and b'_a (empirical) across a sampling of 633 clients on both platforms who used various top hogs at a variety of rates. Carat tended to underestimate the improvement that clients would experience, but 95.4% of these predictions fell within our 95% confidence bounds.

6 Limitations and Future Work

Carat takes a black-box approach to diagnosing anomalies, which carries inherent limitations. Without visibility into the mechanisms (e.g., code, messages, or kernel state) and without the ability to perturb the system (i.e., is passive and cannot modify other apps), the best possible result is to say what aspects of the system are likely to be involved with the abnormal battery discharge. This is what Carat provides, and it does so by correlating real-valued signals from features without initial assumptions about their relationships. This kind of approach has proven fruitful in prior work [25, 27].

Compared to iOS, Android provides greater visibility into the behavior of apps and the operating system, as would facilitating app instrumentation through a developer API. We opted for feature parity with iOS for this paper, in order to evaluate a method that works for both platforms, but plan to leverage such additional data in later versions of the app (and have already begun to do so on the backend).

As with any passive approach, which a regulation iOS app must be, our results are limited by the data. If none of the clients ever runs a particular buggy app, Carat will never detect a problem; if two apps are always run together and one is anomalous, they will both be categorized as anomalies and there is nothing that correlation can do to disambiguate. The likelihood of spurious correlations increases with the number of features (apps and configurations). The way to combat this problem is with more data. For example, as we gather more samples involving highly correlated apps that show one but not the other, we can begin to discern which (or possibly both) are responsible for the anomaly. The results show that our data are sufficient for actionable diagnosis.

7 Related Work

There is a rich body of work in diagnosis for correctness and performance. Recent work identified an emerging class of software misbehavior that afflicts battery life [30] and proposed a method for detecting a specific class of such bugs [32]. We believe our work is the first to automatically detect and diagnose abnormal energy use on mobile devices.

Our approach is a form of statistical debugging, in which (loosely speaking) deviant behavior is called a bug [9]. Such methods have been used to identify code paths correlated with failure [16, 17], concurrency bugs [14], shared influence (surprising behavior that is correlated in time) [25, 27], invariant violation [13], and configuration errors [40]. In the field of security, anomaly-based intrusion detection has a long history [8, 33, 34].

These statistical methods frequently make use of a large number of instances or users of these programs, which is sometimes called a *community*. A recent paper suggested a collaborative debugging framework called MobiBug for mobile devices [1], but they focused on crash problems and dumps, not continuous or intermittent measurements. There is prior work for file systems [41] and peer-to-peer networks [21] that generate alerts based on aggregate behavior.

Projects like the Application Communities project [20] use the community to distribute work; instead, we employ uniform, lightweight instrumentation. There are also security applications for the community besides detection, such as diagnosing problems by discovering root causes [40] and preventing known exploits (e.g., sharing antibodies) [7, 24].

Many projects have sought to profile or emulate energy use on mobile devices [10, 11, 22, 28, 29, 31], sometimes for prediction [36, 39], mitigation [3, 18], or developer tools [15]. Human interface studies have shown that 80% of mobile users will take steps to improve their battery life [35]; Carat recommends specific, personalized actions for users to take and even estimates the benefit they are likely to see. We believe this is a distinguishing feature of our work.

Energy debugging shares similarities with performance debugging; both areas aim to account for the use or abuse of a shared resource. Some notable performance debugging work includes history-based analysis in datacenters [5], resource accounting [4], and blackbox debugging [2]

Pinpoint [6] and Magpie [4] track communication dependencies with the aim of isolating the root cause of misbehavior; they require instrumentation of the application to tag client requests. In order to determine the causal relationships among messages, Project5 [2] and WAP5 [37] use message traces and compute dependency paths. D^{3}S [19] uses binary instrumentation to perform online predicate checks. Recent work shows how access to source code can facilitate tasks like log analysis [42] and distributed diagnosis [12]. CarrierIQ[16] collects detailed measurements by integrating with the mobile platform, and has drawn criticism for the intrusiveness of their implementation[17]. Unlike the preceding methods, we do not assume such access to code, communications, or binaries, taking instead a black-box approach with broader deployment potential.

8 Conclusions

This paper presents a method for diagnosing energy anomalies in the wild given incomplete and noisy instrumentation measurements from a community of clients. We implemented this method as an app for iOS and Android called Carat and deployed it to a community of more than 340,000 users. Carat diagnosed thousands of anomalies, which involves detecting the anomaly, estimating its severity, quantifying the error and confidence bounds on that estimate, and sometimes identifying the device features that are correlated with the anomaly. We also validated our implementation with hardware measurements and synthetic anomaly injection, demonstrating that Carat can accurately estimate energy use and detect anomalies. A collaborative approach is required to diagnose energy anomalies; even complete knowledge of app behavior on a single client could be specific to a configuration or user behavior. We believe this work is the first automatic diagnosis of energy anomalies in the wild, and represents a crucial extension of previous work in distributed and statistical debugging to include a new class of abnormal behavior related to mobile energy use.

Notes

[1]`http://carat.cs.berkeley.edu`
[2]`http://msoon.com/LabEquipment/PowerMonitor/`
[3]`http://www.leydenenergy.com/`
[4]`http://techcrunch.com/2012/06/14/carat-battery/`
[5]`http://bit.ly/yTIUeU`
[6]`http://bit.ly/wsMraK`
[7]`http://bit.ly/WZ4dQi`
[8]`http://bit.ly/QSiv72`
[9]`com.bobisoft.wallpaper.beachatnight`
[10]`com.custom.lwp.FREE_HeartAndLove`
[11]`http://bit.ly/QSixvT`
[12]`http://bit.ly/TLWRhV`
[13]`http://bit.ly/Scgjs2`
[14]`http://gdg.to/xeK9CZ`
[15]`http://bit.ly/ODNyxQ`
[16]`http://www.carrieriq.com/`
[17]`http://onforb.es/zd1zmF`

9 References

[1] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl. There's an app for that, but it doesn't work. Diagnosing mobile applications in the wild. In *HotNets*, 2010.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Methitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[3] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *IMC*, 2009.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[5] P. Bodik, M. Goldszmidt, A. Fox, D. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Eurosys*, 2010.

[6] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *DSN*, June 2002.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP*, 2005.

[8] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *IEEE Symposium on Security and Privacy*, 1992.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, 2001.

[10] D. Ferreira, A. K. Dey, and V. Kostakos. Understanding human-smartphone concerns: A study of battery life. In *Pervasive*, 2011.

[11] J. Flinn and M. Satyanarayanan. PowerScope: a tool for profiling the energy usage of mobile applications. In *WMCSA*, 1999.

[12] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.

[13] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[14] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[15] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. In *HotMetrics*, 2008.

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[18] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao. Energy-accuracy trade-off for continuous mobile device location. In *MobiSys*, 2010.

[19] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: debugging deployed distributed systems. In *NSDI*, 2008.

[20] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *NDSS*, 2005.

[21] D. J. Malan and M. D. Smith. Host-based detection of worms through peer-to-peer cooperation. In *ACM Workshop on Rapid Malcode*, 2005.

[22] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Mobicom*, 2012.

[23] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 1997.

[24] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.

[25] A. J. Oliner and A. Aiken. Online detection of multi-component interactions in production systems. In *DSN*, 2011.

[26] A. J. Oliner, A. P. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica. Collaborative energy debugging for mobile devices. In *HotDep*, 2012.

[27] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *DSN*, 2010.

[28] E. Oliver. The challenges in large-scale smartphone user studies. In *HotPlanet*, 2010.

[29] E. A. Oliver and S. Keshav. An empirical approach to smartphone energy level prediction. In *UbiComp*, 2011.

[30] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *HotNets*, 2011.

[31] 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 *EuroSys*, 2012.

[32] A. Pathak, A. Jindal, Y. C. Hu, and S. Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Mobisys*, 2012.

[33] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Computer Networks*, December 1999.

[34] P. A. Porras and P. G. Neumann. EMERALD: event monitoring enabling responses to anomalous live disturbances. In *NIST/NCSC*, 1997.

[35] A. Rahmati, A. Qian, and L. Zhong. Understanding human-battery interaction on mobile phones. In *MobileHCI*, 2007.

[36] N. Ravi, J. Scott, L. Han, and L. Iftode. Context-aware battery management for mobile phones. In *PerCom*, 2008.

[37] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *WWW*, 2006.

[38] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers—a survey. *IEEE Trans. on Sys., Man, and Cybernetics*, 2005.

[39] A. Sinha and A. P. Chandrakasan. JouleTrack: a web based tool for software energy profiling. In *DAC*, 2001.

[40] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[41] Y. Xie, H. Kim, D. O'Hallaron, M. Reiter, and H. Zhang. Seurat: a pointillist approach to anomaly detection. In *RAID*, 2004.

[42] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[43] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.