Collaborative Energy Debugging for Mobile Devices

Adam J. Oliner[1], Anand Iyer[1], Eemil Lagerspetz[2], Sasu Tarkoma[2], and Ion Stoica[1]
[1]*AMP Lab, UC Berkeley, {oliner,api,istoica}@eecs.berkeley.edu*
[2]*U of Helsinki, {eemil.lagerspetz, sasu.tarkoma}@cs.helsinki.fi*

Abstract

We aim to detect and diagnose code misbehavior that wastes energy, which we call *energy bugs*. This paper describes a method and implementation, called Carat, for performing such diagnosis on mobile devices. Carat takes a collaborative, black-box approach. A non-invasive 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. Carat successfully detected all energy bugs in a controlled experiment and, during a deployment to 883 users, identified 5434 instances of apps exhibiting buggy behavior in the wild.

1 Introduction

Mobile computing, especially in the form of smartphones and tablets, is becoming ubiquitous. Recent work [18] acknowledges the rise of a new class of software misbehavior on these devices: *energy bugs*. These bugs consume energy by performing activities not intrinsic to the app's function. The poor battery life that results from these problems generates frustration among users, poor press for the vendors, and can render devices unusable.

A user of a mobile device with battery problems wants to understand what is depleting the battery, whether that is normal, and what can be done. Any solution for this user must adhere to several hard constraints:

- No hardware modifications. Such solutions are expensive, require technical skill, and void warranties.
- No kernel modifications. Hacking an OS requires skill; even "jailbreaking" may brick the device or introduce 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.

Distribution mechanisms like Apple's App Store and Google's Play Store make it easy to get instrumentation onto off-the-shelf devices, so long as that instrumentation is in the form of a standard app that offers a valuable service. An app that can provide insight into poor battery life, and actionable advice for improving it, clearly meets this requirement.

Unfortunately, a single app instance has limited diagnostic power because not all energy use is energy misuse. There is no *a priori* specification of energy bugs (in contrast to many correctness bugs). The app could measure every local signal—even with kernel or hardware modifications—and still not know whether the observed energy use is normal. The information is simply not present on a single device.

If, instead, we had a *community* of devices, these questions would become tractable. Measurements taken and 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 or not, and predict the expected improvement of taking particular actions.

In this paper, we present a collaborative method for detecting and diagnosing energy problems by looking for deviation from typical battery use (see Section 2) and an implementation as an app for iOS and Android called Carat (see Section 3). Carat uses the community to infer a specification (expected energy use) and flags particular kinds of deviation from that specification. We validate the precision and costs of our method using power metering hardware (see Section 4) and the detection accuracy using synthetic bug injection. Using data from a deployment to 883 users, Carat revealed 5434 instances of apps exhibiting energy bugs in the wild (see Section 5).

2 Method

Our method builds and compares conditional probability distributions of rates of energy use in a *community*; e.g., the rates when an app is running on a *client* with a particular OS version (the *subject distribution*) may be higher than when running on clients with another OS ver-

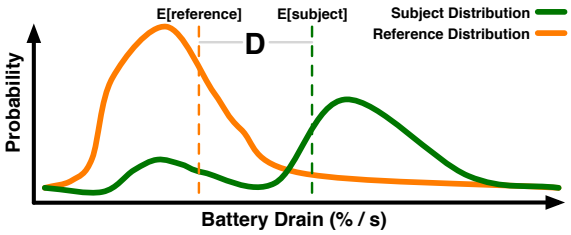

Figure 1: We compare expected values of conditional energy drain rate distributions to classify apps as hogs, bugs, or neither.

sion (the *reference distribution*). We classify behavior by comparing different types of subject and reference distributions. Although we have been investigating more sophisticated machine learning techniques, such as clustering and statistical modeling, this paper focuses on a simpler, yet effective, computation.

2.1 Hogs and Bugs

Informally, an app is an energy *hog* when using that app drains the battery much faster than the average app. An app has an energy *bug* when some running instances of the app (i.e., the ones in which the bug manifests) drain the battery much faster than other instances of the same app (i.e., the ones in which the bug does not manifest).

First, build a (reference) distribution of battery discharge *rates* for a community using devices normally: playing games, making phone calls, leaving it idle, etc. Now, 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 lower than the expected lifetime without A—positive D in Figure 1—we call A an energy *hog*. 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 of the cause, avoiding hogs improves battery life.

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 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 subset of clients, a rare configuration ("correct" or otherwise), or unusual user behavior. 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. As with hogs, this pragmatic definition empowers users to understand and improve the battery life of their device without requiring technical expertise.

2.2 Comparing Rate Distributions

As discussed in Section 2.1, to detect hogs and bugs we compare two distributions of the battery drain (see Figure 1). Let c be the conditions of the subject distribution (e.g., app A is running) and c' be the conditions of the reference distribution (e.g., app A is not running). Let $R(c)$ be the conditional distribution of rates when c holds and $E[R(c)]$ be its expected value. Setting the conditions c (subject) and c' (reference) as described above, we classify the app as a hog or bug if the absolute distance $D = E[R(c)] - E[R(c')]$ (shown) is greater than zero; the relative distance (used in our implementation) is

$$C(c,c') = \frac{E[R(c)] - E[R(c')]}{E[R(c)]} = 1 - \frac{E[R(c')]}{E[R(c)]}. \quad (1)$$

The expected improvement in battery life if the client were to change from c to c' (such as by killing a hog or getting a bug into a normal state) follows directly from this distance metric. The expected value of each distribution ($E[R(c')]$ and $E[R(c)]$) is equivalently the expected rate of energy use, which can be used to predict expected battery life improvement L when moving from one regime to the other (starting from full charge and fully draining the battery): $L = \frac{1}{E[R(c')]} - \frac{1}{E[R(c)]}$. Carat suggests actions that would improve battery life and the expected improvement for an average client.

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 sample is a measurement taken at a particular point in time that consists of the battery level (%) and a list of properties about the client. Let $s_t = (b, p, F)$ denote a sample taken at time t, 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 F of key-value pairs (e.g., "AppXRunning=YES").

First, 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 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 battery levels and timestamps by converting a consecutive pair $s_{t_1} = (b_1, p_1, F_1)$ and $s_{t_2} = (b_2, p_2, F_2)$ to a rate distribution R. If both endpoints, (b_1, t_1) and (b_2, t_2), are exact, then the rate is $r = \frac{b_1 - b_2}{t_2 - t_1}$ with probability 1. Discharging yields a positive rate. 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. On iOS, for example, we use a *prior* distribution computed from more precise measurements to infer the distribution of less precise measurements.

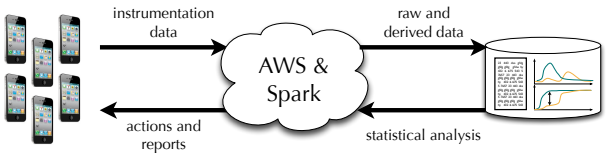

Figure 2: The Carat architecture, showing the crowd-based front end, the central server with the analysis running in the cloud, and the stored samples and results.

We associate this distribution with a set of features F' computed from the features of the constituent pair of samples (F_1 and F_2) by taking the union: $F' = F_1 \cup F_2$. 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 Feature Correlation

To facilitate diagnosis, the analysis looks for correlations between high energy use and features. For example, we might discover that a bug is more likely to manifest on a particular version of the operating system, which in turn might suggest to a developer what code is misbehaving.

We omit the details, but report that feature correlation helped diagnose bugs in the wild, such as on Kindle in Section 5.4 where WhisperSync was using far more energy when syncing over GSM. Our analysis discovered the bug exclusively hit iPads lacking WiFi. Similarly, the Facebook Messenger bug afflicted particular Android device models.

3 Implementation

The Carat architecture consists of an app (see Section 3.1), a central server (see Section 3.2), and an analysis running in the cloud (see Section 3.3). Figure 2 shows an overview of our implementation.

3.1 Carat App

We implemented Carat on the iOS and Android platforms. It is available as a free download on Apple's App Store, Google's Play Store, and on GitHub, all of which are linked from the project homepage[1].

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 state information, using the public APIs, 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 main screen of Carat is the Actions list, which presents personalized actions the user can take to improve battery life, sorted by the expected improvement if that action is taken. For example, it might show an action "Kill Skype" that would result in an expected increase of 47m 21s. This means our analysis observed that a typical device running Skype will run a full battery down to zero 47 minutes sooner than a typical device running typical apps but not Skype. 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, it will suggest upgrading the operating system if it observes that a newer version is correlated, across the community, with better battery life.

The My Device tab shows information about the client's device and 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 95 means a better battery life than 95% of devices. 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.

3.2 Carat Server

The Carat server collects samples from clients running the Carat app and stores them for later use by the analysis engine, and it serves actions and other results of the analysis to clients requesting them. These results include customized action lists, bug lists, and hog lists.

The server is a 1253-line Java application (excluding code auto-generated by Thrift) hosted on Amazon EC2, with mechanisms to scale by spawning new instances and to load-balance incoming connections. The data is stored in Amazon's DynamoDB.

3.3 Backend Analysis

The backend analysis is a 4k-line Scala program also running on EC2. It reads the sample data from the database and performs the calculations described in Section 2.

The analysis converts samples to rate distributions and loads them into Spark RDDs, a distributed data structure that provides caching. Spark is a cluster computing framework that presents a MapReduce-like interface, with support for caching intermediate results [25]. Spark was designed for iterative and incremental computations, which is precisely what our analysis performs. The results—including hogs, bugs, J-Scores, and feature correlations—are stored in DynamoDB where they are retrieved on-demand by the client.

4 Ground Truth and Overhead

In order for Carat to accurately account for when energy is being used without incurring unacceptable overhead, it must convert intermittent (and sometimes low precision) battery level samples into energy drain rates in a way that is faithful to the ground truth. We attached a Monsoon Power Monitor[2] to actual mobile devices (an

iPhone 4S and an LG Optimus 2X) and confirmed that our implementation generates accurate energy distributions while consuming few resources. The methodology and results are similar on both platforms; we present the iOS experiment.

To test the fidelity and cost of sampling, we used a repeatable four-hour script of activities (e.g., browsing the web and idle periods). One of the authors manually ran through the script under three arrangements: (1) hooked to the power meter with and (2) without Carat running and (3) not hooked to the power meter with Carat running. We compare (1) and (2) to quantify the overhead of running Carat; we compare (1) and (3) to ensure the meter was not influencing Carat's measurements and to assess the fidelity of the sampling and rate estimation. For the runs without Carat, where our app appears in the script, we substituted the standard Weather app.

The battery levels reported by the public API, which Carat uses, track the actual use of power by the device, never deviating more than 1.2% from ground truth.

Furthermore, the energy rate distribution computed from the Carat samples using our statistical inference method approximates the distribution computed with the summarized measurements from the power meter hardware (13,549 samples at effectively 0.0001% resolution); using the samples from this four-hour experiment, Carat overestimates the average discharge rate by only 0.0015%/sec.

Carat does not impose significant overhead with respect to energy use, even when taking samples, reporting them to the server, and downloading analysis results. In fact, our power metering hardware indicates that running through our script with Carat running used *less* energy than executing that same script with the Weather app running in its place: 53.691 mAh or $\sim$3.5% of the battery less. Our method can afford to perform such sparse, low-overhead sampling on individual clients because it aggregates data from a large number of clients.

5 Deployment Results

We collected 22,053 samples from 207 iOS devices and 157,323 samples from 676 Android devices, from which our method detected 1159 hogs and 5434 instances of apps exhibiting energy bugs (see Sections 5.2 and 5.4) as well as all 3 energy bugs that we injected into a private deployment of 75 iOS users (see Section 5.3).

5.1 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.

Network traffic scales linearly with the size of our deployment, at a rate far below 1 byte per second per client. A handful of Carat servers could, therefore, support a community composed of every mobile device in the world. Sample reporting is presumed to be unreliable; a client with no disk space or network access is allowed to throw away data, while an overloaded server may drop packets.

Our implementation of the analysis can recompute all results from 181,248 raw samples in 31m 40s on a single machine (8 threads on 4 virtual cores). The computation is massively parallel; each distribution and comparison can be performed independently.

5.2 Hogs

Of the 2664 apps seen during our deployment, 1159 were categorized as hogs. Recall that an app is a hog if the community-wide average discharge rate while running the app is 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 2.2). Hogs may be caused by an oft-triggered code bug but may also be simply intrinsic to the app. We present one example.

Pandora Radio: Carat reports that the Pandora Radio iOS app, which 23 users ran, is a hog and that killing it will increase an average client's battery life by 21m 47s. 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[3]. This is likely an example of a hog behaving as intended: using the radio or WiFi, the speakers, the screen, and other energy-consuming resources. Nevertheless, killing this app is likely to improve battery life.

5.3 Injected Bugs

We added energy bugs to an existing app—initially without apparent misbehavior—to confirm that Carat is able to detect the change. (In Section 5.4, we identify bugs in the wild.) We chose the Wikipedia Mobile iOS app made by Wikimedia Foundation because it was an open-source app being used by many of our clients, but was reported as neither a hog nor a bug. We added several behaviors that consume large amounts of energy when we activate them. These misbehaviors represent a subset of those we expect to see in practice, with each one abusing (repeatedly using) a different resource: radio, CPU, and GPS.

We installed this buggy instance on one of our test devices, an iPhone 3GS. We ran the app for one day for each injected bug, activating the app a handful of times during the day but only leaving it open for a couple of minutes. At the end of the third day, we ran the analysis with the real, non-buggy data from a 75-user iOS deployment as the reference distribution and once each with the data from exactly one of the buggy days as the subject distribution. Indeed, after performing the injection, Carat correctly detected each of the three bugs (no false neg-

atives) and no new bugs were incorrectly reported (no false positives).

5.4 Wild 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). The total number of possible bugs that Carat could report is the Cartesian product of the clients and the non-hog apps. There were 651 client IDs that returned samples during the deployment and there were 1345 apps that were neither hogs nor *daemons* (system processes that users cannot easily terminate). Of the 875,595 possible bugs our method could have reported, the analysis only produced 5434 (for 644 unique apps). In other words, less than 0.62% of the bug subject distributions had a positive distance relative to the reference, meaning that most apps have similar energy rate distributions across devices.

Kindle (iOS): This electronic book app was reported as a bug for 5 of the 11 clients running it. The support forums blame the problem on WhisperSync[4], 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. The forums could provide only anecdotes, whereas our experimental data support this hypothesis; of the 5 iPads that ran Kindle in our deployment, all the GSM iPads reported Kindle as a bug and all the WiFi iPads did not.

Facebook Messenger (Android): This app was a bug for 17 of the 53 clients running it. Certain device models were especially afflicted, particularly the Samsung Galaxy S II (Pearson correlation 0.24). The worst instance reduced battery life by almost six hours. This recalls reports of energy bugs in the main Facebook app reported elsewhere [20] and suggests the extent of the problem in practice.

5.5 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 compute distances—tends to converge to the true value. Figure 3 shows the shrinking relative error of this estimate for six apps seen in our deployment. In practice, the true rate distribution may be neither stationary nor identically distributed, such as in the presence of an energy bug. Carat reported the Facebook iOS app as an energy bug, and, looking back at Figure 3, we can see that app has a bump in the curve.

6 Limitations and Future Work

Carat takes a passive, black-box approach to energy debugging, which carries inherent limitations. Without visibility into the mechanisms (as with invasive methods

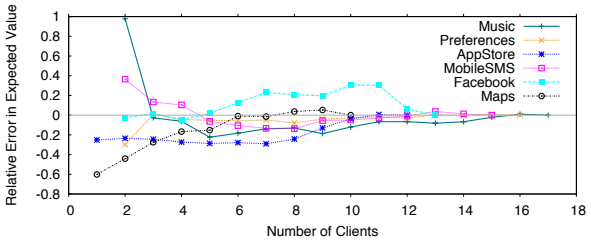

Figure 3: As the number of clients reporting samples increases, the relative error in our estimate of the expected discharge rate shrinks. (iOS data shown.)

like CarrierIQ[5]) and without the ability to perturb the system (i.e., cannot modify other apps), the best possible result is to say what aspects of the system are likely to be involved with the problem. Carat provides this, using as much data as it can access, and it does so by correlating real-valued signals from components with no *a priori* assumptions about their relationships. This kind of approach has proven fruitful in prior work [15, 16].

Our results are limited by the data. If no client ever runs a particular buggy app, Carat will not detect a problem; if two apps are always run together and one is a bug, they will both be categorized as buggy and there is nothing that correlation can do to disambiguate. The likelihood of spurious correlations increases with the number of features (apps and configurations) and decreases with more data; we are working to grow our deployment.

We are building an API that would allow app developers to instrument their code, providing Carat with insight into the states and settings of those apps, as well as how users are interacting with them.

7 Related Work

There is a rich body of work in diagnosis for correctness and performance [2, 4, 5, 9]. Recent work identified an emerging class of software misbehavior that afflicts battery life [18]. We believe ours is the first statistical approach for diagnosing these so-called energy bugs.

Our approach is a form of statistical debugging, in which (loosely speaking) deviant behavior is called a bug [6]. Such methods have been used to identify code paths correlated with failure [13], concurrency bugs [11], shared influence (surprising behavior that is correlated in time) [15, 16], invariant violation [10], and configuration errors [24]. 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.

Many projects have sought to profile energy use on mobile devices [7, 8, 17, 19], sometimes for predic-

tion [22, 23], mitigation [3, 14], or developer tools [12]. Human interface studies have shown that 80% of mobile users will take steps to improve their battery life [21]; Carat recommends specific, personalized actions for users to take and even estimates the benefit they are likely to see. We believe this is one of the distinguishing features of our work.

8 Conclusions

This paper presents a method for diagnosing energy bugs 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 several hundred users. Our method detected thousands of instances of energy misuse, and provided, in several cases, diagnostic information regarding which device models or OS versions the misbehavior most affects. We also validated our implementation with hardware measurements and synthetic bug injection, demonstrating that Carat can accurately estimate energy use and detect bugs. We believe this is the first statistical detection of energy bugs in the wild, and represents a crucial extension of previous work in distributed and statistical debugging to include a new class of misbehavior related to mobile energy use.

Acknowledgements

We thank the folks at Monsoon tech support for helping us connect their hardware to the iPhone, the beta testers and reddit community of /r/compsci for trying early versions of the app, and our colleagues in the AMP Lab for their valuable feedback.

This research is supported in part by NSF CISE Expeditions award CCF-1139158, gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

References

[1] AGARWAL, S., MAHAJAN, R., ZHENG, A., AND BAHL, V. There's an app for that, but it doesn't work. Diagnosing mobile applications in the wild. In *HotNets* (2010).

[2] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND METHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[3] BALASUBRAMANIAN, N., BALASUBRAMANIAN, A., AND VENKATARAMANI, A. Energy consumption in mobile phones: A measurement study and implications for network applications. In *IMC* (2009).

[4] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *OSDI* (2004).

[5] BODIK, P., GOLDSZMIDT, M., FOX, A., WOODARD, D., AND ANDERSEN, H. Fingerprinting the datacenter: Automated classification of performance crises. In *Eurosys* (2010).

[6] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP* (2001).

[7] FERREIRA, D., DEY, A. K., AND KOSTAKOS, V. Understanding human-smartphone concerns: A study of battery life. In *Pervasive* (2011).

[8] FLINN, J., AND SATYANARAYANAN, M. PowerScope: a tool for profiling the energy usage of mobile applications. In *WMCSA* (1999).

[9] GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP* (2009).

[10] HANGAL, S., AND LAM, M. Tracking down software bugs using automatic anomaly detection. In *ICSE* (2002).

[11] JIN, G., THAKUR, A., LIBLIT, B., AND LU, S. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA* (2010).

[12] KANSAL, A., AND ZHAO, F. Fine-grained energy profiling for power-aware application design. In *HotMetrics* (2008).

[13] LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug isolation via remote program sampling. In *PLDI* (2003).

[14] LIN, K., KANSAL, A., LYMBEROPOULOS, D., AND ZHAO, F. Energy-accuracy trade-off for continuous mobile device location. In *MobiSys* (2010).

[15] OLINER, A. J., AND AIKEN, A. Online detection of multi-component interactions in production systems. In *DSN* (2011).

[16] OLINER, A. J., KULKARNI, A. V., AND AIKEN, A. Using correlated surprise to infer shared influence. In *DSN* (2010).

[17] OLIVER, E. A., AND KESHAV, S. An empirical approach to smartphone energy level prediction. In *UbiComp* (2011).

[18] PATHAK, A., HU, Y. C., AND ZHANG, M. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *HotNets* (2011).

[19] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with eprof. In *EuroSys* (2012).

[20] PATHAK, A., JINDAL, A., HU, Y. C., AND MIDKIFF, S. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Mobisys* (2012).

[21] RAHMATI, A., QIAN, A., AND ZHONG, L. Understanding human-battery interaction on mobile phones. In *MobileHCI* (2007).

[22] RAVI, N., SCOTT, J., HAN, L., AND IFTODE, L. Context-aware battery management for mobile phones. In *PerCom* (2008).

[23] SINHA, A., AND CHANDRAKASAN, A. P. JouleTrack: a web based tool for software energy profiling. In *DAC* (2001).

[24] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI* (2004).

[25] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

Notes

[1] `http://carat.cs.berkeley.edu`
[2] `http://msoon.com/LabEquipment/PowerMonitor/`
[3] `http://bit.ly/yTIUeU`
[4] `http://gdg.to/xeK9CZ`
[5] `http://www.carrieriq.com/`