Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study

Mario Linares-Vásquez[1], Gabriele Bavota[2], Carlos Bernal-Cárdenas[3]
Rocco Oliveto[4], Massimiliano Di Penta[2], Denys Poshyvanyk[1]
[1]The College of William and Mary, Williamsburg, VA, USA [2]University of Sannio, Benevento, Italy
[3]Universidad Nacional de Colombia, Bogotá, Colombia [4]University of Molise, Pesche (IS), Italy
mlinarev@cs.wm.edu, gbavota@unisannio.it, cebernalc@unal.edu.co,
rocco.oliveto@unimol.it, dipenta@unisannio.it, denys@cs.wm.edu

ABSTRACT

Energy consumption of mobile applications is nowadays a hot topic, given the widespread use of mobile devices. The high demand for features and improved user experience, given the available powerful hardware, tend to increase the apps' energy consumption. However, excessive energy consumption in mobile apps could also be a consequence of energy greedy hardware, bad programming practices, or particular API usage patterns. We present the largest to date quantitative and qualitative empirical investigation into the categories of API calls and usage patterns that—in the context of the Android development framework—exhibit particularly high energy consumption profiles. By using a hardware power monitor, we measure energy consumption of method calls when executing typical usage scenarios in 55 mobile apps from different domains. Based on the collected data, we mine and analyze energy-greedy APIs and usage patterns. We zoom in and discuss the cases where either the anomalous energy consumption is unavoidable or where it is due to suboptimal usage or choice of APIs. Finally, we synthesize our findings into actionable knowledge and recipes for developers on how to reduce energy consumption while using certain categories of Android APIs and patterns.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement

Keywords

Energy consumption, Mobile applications, Empirical Study

1. INTRODUCTION

In recent years, we are observing rapid evolution of mobile devices in terms of available hardware, operating systems, and, as a consequence of that, the growing lists of features that mobile applications' (apps) users demand. These modern apps have virtually the same features as their equivalent desktop applications. For instance, many top video games for mobile devices provide similar levels of user experience as compared to those console analogs. Such evident step-ahead has, however, a price to be paid. Nowadays, multi-core processors, high-performance Graphical Processing Units (GPUs), and large screens on mobile devices are becoming more energy demanding as ever. Also, apps fully exploiting available hardware can easily drain devices' batteries in no time.

From a user's perspective, this produces tangible and pertinent problems. The use of energy-draining apps could quickly leave a user with empty battery, preventing her from using the smartphone even for phone calls. In addition, having and running such apps might require frequent battery re-charges. This represents a problem because modern battery's life is quite limited, often to a finite amount of charging cycles (for Lithium-ion batteries), ranging between 300 and 500 cycles (with only 100-200 cycles after a mid-life point) and gradually decreasing with time [4, 5].

A practical, although naïve advice for preventing rapid discharges and for improving batteries' life, is to use mobile devices only for low energy consuming scenarios. However, while it might be obvious that some apps are likely to be power demanding—e.g., video games or those apps using devices such as Global Positioning Systems (GPS)—it can often happen that some apps might quickly drain the battery without any apparent reason [32, 33]. For instance, several studies identified misuses of *wakelocks* that keep hardware components unnecessarily awake as causes of high energy consumption in mobile devices [22, 32, 33, 35].

Also, programming errors, hardware interactions, and API misuses can cause high levels of energy consumption (also known as *energy bugs*) in mobile apps [32]. To identify such problems, effective strategies for measuring energy consumption in mobile devices are needed. In the literature, several different strategies have been proposed, based on real measurements [6, 11, 21, 23, 25, 40] and power modeling [19, 20, 33, 34, 43, 46]. While previous work attempted at characterizing energy bugs in mobile devices [6, 7, 19, 33, 40], most of these classifications have been done either by mining software repositories (e.g., bug reports, forums, commit logs)

[32, 42, 44] or by using dynamic tainting [27, 45]. Thus, there is a clear gap in the research literature on how and where the uses and misuses of APIs can lead to energy bugs based on large-scale empirical data. Up-to-date, only the *wakelock* and *GPS* related APIs and their misuses have been studied and linked to energy bugs [34, 35, 42, 44].

Based on these considerations, our goal is to conduct a quantitative and, above all, qualitative exploration into how different API usage patterns can influence energy consumption in mobile apps. We mined and analyzed thousands of instances of energy-greedy method calls and API usage patterns by measuring their energy consumption in 55 free Android apps belonging to different domain categories. In order to collect energy consumption values for each API call, we used a hardware-based approach for collecting actual energy measurements and aligned those values with execution traces generated from real usage scenarios on mobile apps. Once energy consumption data were collected for all execution scenarios, we traced all API calls back to the source code where they appear using an approach that we specifically implemented in this paper. Note that, since previous studies had already shown that 3G/GSM and GPS were energy-greedy hardware components [6, 19, 33, 34], in this study we are not interested in measuring the energy consumed by APIs related to those components.

Using the proposed measurement framework, we analyzed the consumption of individual APIs as well as their usage patterns. Firstly, we quantitatively identified the APIs and the patterns exhibiting high energy consumption. Then, several evaluators inspected—following a categorization approach inspired from the grounded theory [12]—thousands of code fragments where such APIs/patterns occurred. In some cases, we found evidence of energy bugs due to API misuses or suboptimal API choices. Overall, our work is in the trend of mining energy consumption patterns for software systems, originally pioneered in this community by the work of Hindle [21]. The contributions of this paper are:

1. An approach for identifying API calls and usage patterns exhibiting high energy consumption;
2. Quantitative ranking of high energy-demanding APIs and usage patterns, which have been traced back to original source code in the apps;
3. Results of a qualitative analysis aimed at understanding energy hotspots in API usage patterns and discussing whether alternative solutions are available;
4. Actionable knowledge and recipes for developers on how to reduce energy consumption while using certain categories of Android APIs and patterns.

Replication package. All the data used in our study are publicly available online [26]. In particular, we provide (i) the list of the apps and their versions, (ii) the scenarios we used for executing the apps, and (iii) the raw data of energy consumption measurements.

2. EMPIRICAL STUDY DESIGN

The *goal* of this study is to investigate energy-greedy Android APIs, with the *purpose* of understanding particular instances of API calls and API usage patterns that cause (unusually) high energy consumption. The *quality focus* is on the identification of API usages that can cause battery draining and, if any, possible energy bugs.

Table 1: Distribution of 55 apps across categories.

Category	Apps (%)	Category	Apps (%)
Tools	15(27.27%)	Communication	2(3.64%)
Music	6(10.91%)	Entertainment	2(3.64%)
Books	4(7.27%)	Health	2(3.64%)
Productivity	4(7.27%)	Brain	1(1.82%)
Arcade	3(5.45%)	Business	1(1.82%)
Media	3(5.45%)	Casual	1(1.82%)
News	3(5.45%)	Education	1(1.82%)
Travel	3(5.45%)	Finance	1(1.82%)
Cards	2(3.64%)	Lifestyle	1(1.82%)

The *context* consists of *software*, i.e., a set of free Android apps from Google play, and *hardware*, i.e., an Android-based smartphone. Specifically, we considered 55 free Android apps, belonging to different domains (see Table 1). As for the hardware, we used a brand new, unlocked and rooted Nexus 4 [18] LG phone with a 1.5 GHz quad-core Qualcomm Snapdragon S4 Pro CPU, and equipped with Android 4.2 (kernel version 3.4.perf-ge039dcb). The choice of an Android-based environment (i.e., device, OS, and apps) is not random but motivated by two main reasons. First, the apps under study can be downloaded freely from the market and, thus, our results can be fully reproduced. Second, the Android framework tools allow remote execution of apps (from a laptop connected to the device) for debugging and profiling purposes.

2.1 Research Questions

Our study aims at empirically answering the following research questions (RQs):

- **RQ**$_1$: *Which are the most energy-greedy Android API methods?* This research question focuses on individual API method calls. The goal is to understand whether or not there are any particular API calls consuming more energy than others.
- **RQ**$_2$: *Which sequences of Android API calls are the most energy-greedy?* This research question focuses, instead, on patterns that are composed by subsequent invocations of API calls, possibly interleaved with invocations of other Java methods, i.e., JDK or external libraries. Some of these patterns reflect specific usage scenarios of certain APIs; for example, querying a database implies preparing an SQL statement, executing a query, and analyzing the result set. In this study, we consider patterns with length of two and three.

The **dependent variable** considered in this study is the energy consumption—expressed in Joule (J)—of each method call (**RQ**$_1$) or of each pattern (**RQ**$_2$). In addition, to have a practical implication of the results achieved, we compute the percentage of the battery charge consumed by a method call or pattern consuming a certain amount of Joules. In our context, the Nexus 4 is equipped with a 2,100 mAh, 3.8V battery. Thus, a method call consuming 0.01 J will consume $3 \cdot 10^{-5}$% of the total battery charge:

$$100 \cdot \frac{0.01}{3.8} \cdot \frac{1,000}{2,100 \cdot 3,600} = 3 \cdot 10^{-5}\%$$

The dimensional analysis of the formula is $\frac{[V][I][S]}{[V]} \cdot \frac{1}{[I]\cdot[S]}$

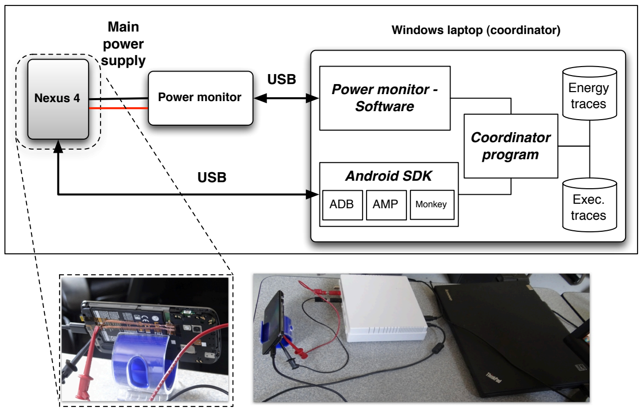

Figure 1: Energy trace collection infrastructure: general architecture (top), phone-to-power monitor connection (bottom left), test bed (bottom right).

where $[V]$ indicates voltage, $[I]$ current intensity, and $[S]$ time. As shown, the formula provides a dimensionless quantity, indicating the percentage of battery discharge. Note that 1,000 at the numerator is because the charge is in mAh and not Ah, and 3,600 is to convert hours to seconds.

Since this is mainly a qualitative study, we are not focusing on identifying independent variables to control and correlate with the dependent variable. Instead, as it will be explained in Section 2.4, we inspected occurrences of method calls and patterns in the source code and categorized them using grounded theory approach [12].

2.2 Defining the App Execution Scenario

We defined and executed representative scenarios to measure energy consumption for methods and patterns for each of the 55 considered apps. Rather than collecting execution traces with the objective of maximizing code coverage, we focused on typical usage scenarios, which may not have necessarily resulted in the highest levels of coverage possible. For example, for a media player one of the execution scenarios consisted of creating a play list and then playing its songs for ten seconds each, while for a browser execution scenarios involved surfing the Web, managing bookmarks and such, and for a travel guide scenarios such as looking for places in a city and transportation options. The scenarios have then been recorded in test scripts by using the `Monkey Recorder` tool [1], so that their execution can be automated. This was needed because each scenario was executed 30 times and the consumption of each method was averaged to minimize the measurement randomness (e.g., due to external factors, such as other running processes).

2.3 Data Extraction Process

This section explains in detail the data-gathering for estimating the energy consumed by API method calls. The data extraction process follows a three-step approach. In the first step, we monitored energy consumption executing usage scenarios and at the same time collecting execution traces. In the second step, we aligned execution traces with energy measurements. Finally, in the third step, we traced back the API calls and patterns to source code.

2.3.1 *Power Measurement and Profiling*

Figure 1 depicts the test bed components and their interactions. For collecting execution traces, we used the Android Activity Manager Profiler (AMP) [29]. Note that running the Android AMP requires the apps to be enabled for debugging, and for this reason we had to enable it by modifying the manifest file for some of the apps. Because we had to decompile the APK (Android PacKage) files containing the apps to an intermediate representation after modifying the manifest, we recompiled and signed the files to have valid APKs.

Turning to the energy profiling, we utilized the Monsoon power monitor [29]. We decided to use a hardware-based measurement instead of power modeling [19, 20, 33, 34, 43, 46] because using the power monitor allows us to measure exactly the energy consumed by the device. The whole data collection process was executed from a laptop (here in after referred to as a *coordinator*) connected to the phone via USB. This made necessary to disable the USB charging to avoid any bias in the resulting measurements. To limit noise in the measurement, (i) we disabled all the unnecessary apps and processes running on the phone to avoid race conditions; (ii) we put the phone in airplane mode to avoid cell radio power consumption and asynchronous events related to incoming messages or calls (as recommended in the Android developer guide [16]); (iii) we turned the WiFi on because some apps require it; and (iv) to avoid energy measurements by sensors we held the phone steady. We are aware that such a setting prevented us to study GPS and 3G/GSM related APIs. However, previous work has shown that GPS and 3G/GSM are energy-greedy hardware components [6, 19, 33, 34]. We are not interested in such obvious cases of energy-greedy APIs, therefore, the goal of this study was to study previously unexplored APIs and patterns.

We summarize all the energy data collections steps in Listing 1. The entire collection procedure going from the app installation to the trace generation was entirely automated and managed by the *coordinator*, by executing the test scripts produced as explained in Section 2.2. For each execution, we installed/uninstalled the app under test remotely to avoid unexpected behavior because of previous data or application's state. After each app was installed, the execution was started and the AMP was attached to the process running the app[1]. Moreover, as explained, each test script was executed 30 times. This procedure was not applicable only to four apps (i.e., Angry Birds Star Wars, Sniper Shooter, Despicable Me Minion Rush, and Arcane Legends), since those apps required gesture events and had a non-deterministic behavior. For these apps, we programmed the *coordinator* to collect data ten times for five minutes without using test scripts; during the collection period one of the authors played the games. At the end of each execution, the files generated by the AMP were pulled from the phone to the *coordinator*.

2.3.2 *Aligning Traces and Energy Measurements*

After collecting all the data, we analyzed execution and energy traces offline to assign energy-related measures (i.e., voltage, electric current, and energy) to public Android API methods. The files generated by the AMP are in the binary format that can be read using the Android `Traceview` tool. However, `Traceview` is only a visualizer and does not export the traces to the textual format. Thus, we translated execu-

[1]In this way we collected the execution trace generated only from the app under test.

Listing 1 Energy Data Collection Procedure.

```
Setup(app)
  InstallInPhone(app)
  StartPowerMonitor()
  StartProfiler(app)
end
FinishExecution(app)
  StopProfiler(app)
  StopPowerMonitor()
  Uninstall(app)
  PullProfilerData(app)
end
CollectData(Apps, Scripts, Automatic, Manual)
for all  $app \in Apps$ do
  if $app \in Automatic$ then
    for  $iteration = 1$ to 30  do
      Setup(app)
      $script$ = Scripts[app]
      Execute(script)
      FinishExec(app)
    end for
  else if $app \in Manual$ then
    for  $iteration = 1$ to 10  do
      Setup(app)
      WaitFor(5) //minutes
      FinishExec(app)
    end for
  end if
end for
end
```

tion traces into plain files using the `dmtracedump` tool [17]. A description of the traces is provided in our online appendix.

In the case of energy traces no pre-processing was required because the APIs of the Monsoon power monitor allowed us to get voltage [V] and current intensity [I] values directly, in Volts and milli-Amperes (mA), respectively. However, the sampling frequency of the power monitor is 5KHz, which means that an energy trace is a sequence of time slots with duration of 200 μs. Thus, because of the sampling difference between energy and execution traces, we used an approach similar to the one used by Li *et al.* [25]. Let T be an energy trace with an initial timestamp $t_{T,0}$ and a final timestamp $t_{T,n}$, and S a set of time partitions (time slots) on T with a length of $\Delta t = 200\ \mu s$. The energy consumed by the phone—measured in Joules—at one slot $s \in S$ is:

$$E(s) = V(s) \cdot I(s) \cdot 10^{-3} \cdot \Delta t \cdot 10^{-6} = 2 \cdot V(s) \cdot I(s) \cdot 10^{-7} \quad (1)$$

Consequently, because an API method can be executed in a time period that is included in an individual time slot s or in a time period that is extended through more than one time slot, the energy consumed by the i-th execution of a specific method $m^{(i)}$ (e.g., the fifth call to the method `Context.getSystemService`) in a trace T is equal to the sum of the energy consumed during each time slot in T where the execution is done:

$$E_{m^{(i)}}(T) = \sum_{s \,\in\, slots(m^{(i)},T)} E_{m^{(i)},T}(s) \quad (2)$$

This approach assumes that the energy consumed by a specific method execution in a time slot is proportional to the execution time in the slot [25]. Moreover, if several methods are executed at the same time, the energy consumed is inversely proportional to the number of methods executed at the same moment. Thus, given $[t_a, t_b]$ the execution period of a method $m^{(i)}$ in a time slot s, and $methodsAt(t,T)$ the set of methods executed in parallel at time t in the trace T, then the energy consumed by the specific method execution $m^{(i)}$ of trace T in the time slot s is defined as follows:

$$E_{m^{(i)},T}(s) = \frac{E(s)}{\Delta t} \sum_{t \,\in\, [t_a,t_b]} \frac{1}{methodsAt(t,T)} \quad (3)$$

After we estimated the energy for each method execution in a trace (i.e., $E_{m^{(i)}}(T)$) using equation 2, for each trace, we generated a file with the API calls, entry and exit timestamps, nesting level, and the energy consumed in the execution. We ordered the lines in those files using the same order of execution in the original trace.

Once API method calls were collected together with their energy consumption estimates, we computed API usage patterns. A pattern of length n is composed of n method calls and its energy consumption is the total energy consumption of the n invoked methods. As mentioned before, the AMP collects all the calls to Java and Android API methods including public, private, and protected ones that were invoked by the threads managed by the process running an app. However, we were only interested in patterns composed of calls to public Android methods. For this reason, we removed Android non public methods from the identified patterns. This means that private and protected methods are considered when computing energy consumption but they are not included in the identified patterns. We focused on patterns with length between one (i.e., single method calls) and three (i.e., sequences of three calls to API methods)

2.3.3 Tracing API usages to Source Code

Collecting execution traces using the AMP allows us to identify the Android API methods invoked when executing the apps without instrumenting the source code. However, this has a drawback: The traces do not include the app methods responsible for API invocations. In addition, we removed the calls to Java APIs. Thus, some of the sequences of API calls extracted from the traces could not belong to the same scope/context and could not be sequentially invoked in the apps. In order to solve these issues and to understand the purpose of each pattern, we traced-back the identified usage patterns (those with length higher than one) onto the source code methods of actual apps.

Our approach is based on the concept of fingerprint/signature analysis that has been previously used for detecting file cloning [28, 31], and software provenance [9, 10]. Such approach represents methods in a class by means of a fingerprint, defined as the sequence of signatures of the invoked API methods. These fingerprints are then matched in the apps source code.

2.4 Data Analysis Method

For both $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we report quantitative data and qualitative results. In terms of quantitative analysis, we report the distribution of the energy consumption for the executed method calls and for all the identified patterns. Such quantitative analysis is preliminary to the qualitative one, since it allows us to identify the methods and patterns that consume more energy and that, at the same time, are frequently used in Android apps.

Specifically, in our quantitative analysis we focused on individual (**RQ**$_1$) and sequences (**RQ**$_2$) of API calls having a consumption greater than $Q_3 + (1.5 \cdot IQR)$, where Q_3 is the third quartile of the energy consumption distribution, and IQR is the inter quartile range. Using such an energy consumption threshold, we identified a set of 131 API method calls, 15 patterns of length two, and eight patterns of length three. For each of these APIs and API usage patterns we manually inspected their occurrences in the source code of all the apps aiming at deriving the following information:

1. Its *purpose*, e.g., executing a query on a database, handling a socket connection, instancing a secure connection, etc.
2. A likely, preliminary *category* to which the pattern/API method could be assigned. Such a categorization was performed following the procedures from the grounded theory [12], i.e., by relying on keywords found in the source code, domain analysis, and experts' (personal) knowledge.
3. Wherever possible, we inspected specific *forums* (e.g., programmingforums.org), Questions & Answers websites (e.g., StackOverflow), and official Android documentation, to investigate whether such patterns actually represent cases of energy bugs, wrong API choices, or whether they are just cases one has to live with.
4. Also, whenever possible, we investigated and discussed whether there exists a *possible alternative* to the APIs under analysis that could lead towards lower energy consumption.

After this initial categorization, the categories were consolidated, trying to reduce overlaps and small/singleton categories. The final result is a categorized list of energy-greedy patterns or API calls, with possible indication of the presence of energy bugs and suggestion of possible alternatives and/or countermeasures.

3. STUDY RESULTS

This section reports the results aimed at answering the research questions formulated in Section 2.1.

3.1 Analysis of Android API Methods

Figure 2(a) represents the distribution of energy consumption values for the 807 Android API methods that were exercised in the 55 subject apps. Note that the x-axis is reported in log scale for the sake of readability. A distribution fitting performed using the *R* [36] `power.low.fit` procedure of the *igraph* package, indicates that the distribution of energy consumption fits a *power law*, i.e., a function of type $f(x) = \alpha \cdot x^k$. The obtained power law exponent (k) is -1.69, while the Kolmogorov-Smirnov test [8] p-values returned by the fitting procedure is 0.76. Since it is greater than 0.05, this indicates that with a significance level of 5% we cannot reject the hypothesis that the observed distributions deviate from the power law.

Out of all the analyzed APIs, 131 represent negative outliers in terms of energy consumption (see Section 2.4). While the average energy consumption for the remaining 676 (807-131) methods is 4e-5 J on average, for the top-131 methods it is 5e-3 J on average (125 times higher) with peaks of 0.15 Joule, i.e., more than 3,000 times than the average of all the methods. Note also that these API invocations are scattered across all the 55 analyzed apps, with a total of 9,609 instances (mean 73, median 23).

Table 2: Distribution of 131 energy-greedy APIs.

Category	# API Methods (%)
GUI & Image Manipulation	49 (37%)
Database	30 (23%)
Activity & Context	17 (13%)
Services	13 (10%)
Web	7 (5%)
Media & Animation	5 (4%)
Data Structure Manipulation	3 (2%)
File Manipulation	3 (2%)
Geo Location	2 (2%)
Networking	2 (2%)

As designed, our qualitative analysis focuses on the analysis of these 131 API methods to understand whether they represent a wrong implementing choice made by developers (i.e., energy bugs) or just cases where developers had no different choice to implement the required feature. Table 2 reports the categories in which these 131 methods have been classified. For the sake of space, we discuss only the most interesting examples, while the complete list of APIs and energy measurements can be found in the online appendix.

As we can see, APIs related to *GUI & Image Manipulation* and *Database* represent, all together, 60% of the energy-greedy APIs. Concerning the *GUI & Image Manipulation* category, an interesting case is represented by the method `notifyDataSetChanged` of class `ArrayAdapter`. In the Android documentation, this method is described as the one in charge of notifying the attached observers that the underlying data has been changed and any `View` reflecting the data set should refresh itself. This method is energy-greedy due to the necessity of refreshing all views when changes happen to the data represented in them. This method appears to be a well-known energy bottleneck and it has been spotted and discussed by Android developers on Stack Overflow [37]. However, this example represents one of those cases where developers have no choice to implement automatic updating of view basic blocks in response to data changes.

Several methods from the *Database* category are needed in the management of an SQLite database on Android. We go from the database opening `SQLiteDatabase.openDatabase`, to its querying `SQLiteDatabase.query`, until its deletion `ContextWrapper.deleteDatabase`, for a total of 30 energy-greedy APIs. It should be noted that energy-greedy APIs responsible for XML files manipulation are, instead, very rare, with only the method `XML.newSerializer` belonging from the *File Manipulation* category (see Table 2). Of course, this does not mean that the use of an SQLite database represents an energy bug, but just that developers should seriously think if their application really needs to use a relational database as a storage layer or, instead, could also store persistent data in XML files or using `SharedPreferences` [14] for storing key-value pairs of primitive data types.

An interesting case of an energy bug was observed, surprisingly, in the *Data Structure Manipulation* category. Among the energy-greedy methods falling in this category, one was a very simple getter method (i.e., `Bitmap.getPixel(int,int)`) in charge of retrieving a specific pixel from a matrix of integer representing a `Bitmap` image. While this could seem as a very simple operation, we found this particular operation to be very expensive in terms of energy consumption. When investigating this in depth, we found that this specific method was discussed on Stack Overflow [38] by Android developers as a very slow (i.e., computationally expensive) method. In the discussion, one of the participants explained that:

For functions as simple as `setPixel`*,* `getPixel`*, the function call overhead is relatively large. It would be a lot faster to access the pixels array directly instead of through these functions. Of course that means you have to make pixels public, which is not very nice from a design point of view, but if you absolutely need all the performance you can get, this is the way to go.*

Also the official Android API documentation performance tips [13] suggest using getters and setters when accessing internal class fields *could be a bad idea on Android*, since *virtual method calls are expensive, much more so than instance field lookups.* This highlights the fact that good programming practices (information hiding in this case) should not always be adopted while programming for mobile devices, where energy savings is often the priority when building an efficient app. Note that we found usage of the `getPixel` method in 27 of the 55 analyzed apps, highlighting that it is quite popular in our sample of apps.

In the *Web* category, which included all the APIs related to internet surfing, we found seven methods having high energy consumption. Among those, we found constructors of `WebView` class to be especially energy-greedy. While investigating those methods, we found that developers experience troubles to stop the `WebView` thread when the user switches out of it. In fact, in order to do this the code should invoke the `onPause`/`onResume` methods on the `WebView`. However, these methods are hidden and the Java reflection mechanism is needed to access them [39]. Thus, the design of the `WebView` in Android often pushes the developers to inadvertently introduce this not-so-obvious energy bug.

Other instances of energy-greedy APIs are, for example, the `Context.bindService` method (category *Service*), in charge of connecting an application to a bound service, send requests and receive responses, or the `Activity.findViewById` (category *Activity* and *Context*). While we found many other instances of energy-greedy APIs in the remaining categories (e.g., *Media and Animation*, *Geo Location*, *Networking*), these energy problems are rather self-explanatory and somewhat expected.

3.2 Analysis of Android API Usage Patterns

Figures 2(b) and 2(c) show the distribution of energy consumption values for Android API usages patterns with two and three calls. Also in this case the x-axis is reported in log scale for the sake of readability and the distribution of energy consumption fits a *power law.* The obtained power law exponent (k) is -1.53 (size two) and -1.55 (size 3), while the Kolmogorov-Smirnov test [8] p-values returned by the fitting procedure are 0.37 and 0.60, respectively. Again, we cannot reject the hypothesis that the observed distributions deviate from the power law.

We found 642 and 319 different patterns with length two and three respectively in the considered set of apps. For the former (length = 2) there were 15 negative outliers, while for the latter (length = 3) we found eight outliers.

Table 3 reports the categories in which the energy greedy patterns have been classified. As already observed when analyzing single API methods, also in this case energy-greedy patterns mainly fall into two categories: *GUI Manipulation* and *Database.* Note that this holds for both patterns of length two and three. As in Section 3.1, for the sake of space, we discuss in the following only the most interesting examples, while the complete list of patterns and energy measurements can be found in the online appendix. Also, code snippets detailing the most interesting patterns are in our online appendix[2].

Table 3: Distribution of energy-greedy API patterns.

Category	# Patterns l = 2 (%)	# Patterns l = 3 (%)
GUI Manipulation	8 (53%)	5 (62%)
Database	5 (33%)	3 (38%)
Web	1 (7%)	0 (0%)
Activity & Context	1 (7%)	0 (0%)

The pattern `<Activity.setContentView(int); Activity.findViewById(int); View.setVisibility(int)>` is the most energy-greedy sequence we found, with an average consumption of 0.20 Joules. It is used for setting the content view of an `Activity` and make it visible to the user. Most of the energy consumption for this pattern is due to the `Activity.findViewById(int)` method (also present among the 131 greedy-energy methods identified in $\mathbf{RQ}_1$). This method is in charge of finding a `View` basic block identified by the *id* passed as a parameter. The problem is that the views structure for an Android app is stored in XML files (known as layout files) that can easily reach very large size. Thus, in order to find a specific view given its *id*, all the layout files must be iterated through, which is a computationally expensive operation resulting in high-energy consumption as well. Although the consumed energy depends on the amount of views declared in the layout files, some developers recommend to save a global private instance of every visual component that will be used instead of calling `Activity.findViewById` often, as described by an Android Framework Engineer in the Google Forum [2]:

How expensive findViewById? [...] Actually it's not nearly so smart – it is just a traversal through the view hierarchy until it finds a matching id [...] As with all things, you should avoid doing this repeatedly if you don't need to (keep the thing you find in a variable so you don't have to look it up again).

Curiously, this is exactly the opposite of what we found in several apps, like a birthday reminder app showing an app method with more than 50 calls to the `Activity.findViewById(int)` API. We looked for the energy consumed by those calls and we found that the 57 executions of `findViewById` can consume up to 0.22 J, which represents $8 \cdot 10^{-3}\%$ of the battery of a Nexus 4 smart phone. Note that this is the consumption caused by the execution of just one method, and that is given the fact that multicore CPUs implemented in modern smart phones are able to execute millions of methods in less than a second time.

Among the energy-greedy patterns related to database operations, interesting ones are those represented by sequences of calls to the `SQLiteDatabase.execSQL(String)` method, especially long when the statements are used to create/drop database elements. We found some examples, such as one

[2]`http://bit.ly/1fCsjwz`

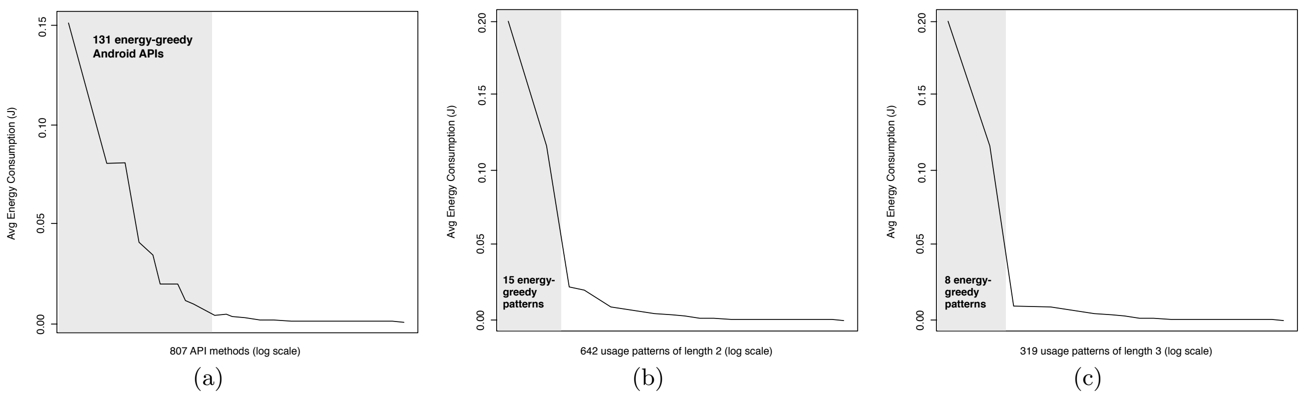

Figure 2: Energy consumption (in Joule) of 807 Android APIs invoked in 55 apps, (b) API usage patterns of length two, and (c) of length three.

```
public void open() throws SQLiteException {
    try{
        this.db = this.dbHelper.getWritableDatabase();
        return;
    } catch (SQLiteException){
        this.db = this.dbHelper.getReadableDatabase();
    }
}
```

Figure 3: Usage example of the pattern `<SQLiteOpenHelper.getWritableDatabase(); SQLiteOpenHelper.getReadableDatabase()>` in a *Productivity* app.

method in an *Education* app in which 26 `execSQL(String)` invocations are performed, leading to battery consumption up to $3 \cdot 10^{-3}$% for a single execution of method.

An interesting example of energy-greedy pattern related to database operations, is `<SQLiteOpenHelper.getWritableDatabase(); SQLiteOpenHelper.getReadableDatabase()>` (see Figure 3), with an average consumption of 0.16 J. Those two methods are used to get access to the database, however one provides access to the database in reading/writing mode (i.e., `getWritableDatabase()`) while the other one just provides access to the database in reading mode (i.e., `getReadableDatabase()`). It is important to highlight that this pattern is quite rare (just two instances found), but there is a reason for this. In fact, these two methods should not be used together. We found an example in a *Productivity* app, in which programmers are managing possible problems occurred when opening the database in reading/writing (e.g., the smart phone memory is full) by opening the database just in reading mode and avoiding the thrown of an `SQLiteException`. In a correct behavior the exception should be thrown to alert the invoker of method `open()` of such a problem.

Another example is an instance of `<ConnectivityManager.getNetworkInfo(int); ConnectivityManager.getNetworkInfo(int); NetworkInfo.isConnected()>` in a *News and Magazines* app. It is used to monitor available network connection; in this example, two types of network are monitored. The pattern consumes on average 0.003 J, and we found that it can be implemented with a call to the method `ConnectivityManager.getActiveNetworkInfo()` instead of using `ConnectivityManager.getNetworkInfo(int)` for each type of network [15].

Also, two widgets that are often used in Android apps are members of energy-greedy patterns: `ProgressBar` and `Toast`. The former is used for visualizing the state and remaining time of activities such as downloads, upgrade, installation, etc.; the latter is used for displaying small popups without hiding or blocking the current activity. An instance of `<ProgressBar.setProgress(int); ProgressBar.setProgress(int)>` on average consumes 0.007 J. However, frequently updating a `ProgressBar` in the main method of a `Runnable` object during a long task could consume a considerable amount of energy. In the case of `Toast`, although it is a transient widget, the pattern `<Toast.makeText(Context, CharSequence,int); Toast.show()>`, which is used to create and show a small non-modal popup in a web browser, consumes 0.008 J on average.

Finally, in the Web category we found patterns including the creation of a `Webview` as energy-greedy ones. As previously explained in the context of our $\mathbf{RQ}_1$ this is mainly due to the fact that developers experience troubles in stopping the `WebView` thread when the user switches out of it.

Based on the obtained results, we derive pieces of *actionable knowledge* or *Energy-Saving Recipes (ESR) for Android developers* reported in Table 4. Each *ESR* derives from one or more examples, and is intended to provide practical advice to both developers, who are interested in avoiding energy bugs, and researchers, who are interesting in deriving recommender systems aiming at guiding developers in building energy green apps.

4. THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation, and are essentially due to the measurements/estimates on which our study is based. In this work our main goal was to have accurate energy consumption measures, and for this reason we have used a hardware meter instead of simulators or available energy profiles. To measure the consumption of each method we had to compute a proportion over the time frame for which the meter provided the measure. Although this is clearly an approximation, it has been performed in a related study [25], which however includes in the approximation tail energy and we do not; thus, it is also a possible threat. Another threat is due to

Table 4: Actionable knowledge: energy-saving recipes (ESRs) for Android developers.

ESR_1	**Carefully design the storage strategy**. Using DBMS is expensive for managing persistence data and should only be used when forced. In addition, creating the schema is particularly expensive since it requires the execution of `SQLiteDatabase.execSQL(String)` multiple times. Since schema creation is performed at the first usage of the App, it could make the user less confident about the efficiency of the app.
ESR_2	**Limit the use of the Model-View-Controller (MVC) pattern, especially when used in apps with many views**. Refreshing views is expensive. Thus, avoid the use of MVC, unless there are no alternatives. Although it increases coupling, an explicit polling may reduce energy consumption. In addition, avoid unnecessary refreshing, i.e., refresh operations made on inactive or invisible views.
ESR_3	**Limit the use of energy-greedy widgets for cyclic activities**. Updating the state of widgets or calling energy-greedy methods inside loops or in the main method (`Runnable.run()`) or `Runnable` objects could be an expensive operation when considering the energy consumed as a result of repetitive actions.
ESR_4	**Carefully design apps that make use of several views**. The navigation of `View` components is energy consuming. Navigating and identifying `View` components is energy expensive since it is necessary to browse the layout files in Android. This calls for an alternative solution for facilitating the identification and browsing of views.
ESR_5	**Carefully analyze the trade-off between design principles and battery saving**. This trade-off should be particularly analyzed, since design principles could be energy-greedy. For instance, information hiding in Android is quite expensive. Thus, disregarding such a principle (giving direct access to private field) could actually help to save battery.

how patterns have been matched in the analyzed apps, i.e., by matching method calls in the source code. Instrumenting apps and tracing method entry and exit points would have avoided this problem. However, we decided not to do so to minimize the noise to the measurements. In the end, we preferred to loose some pattern instances rather than having unreliable measures. Since this is mainly a qualitative study, we are interested to discuss relevant examples rather than having a full coverage of patterns.

Threats to *internal validity* concern factors that can affect our results. Also in this case, such threats can arise during the measurement process. As explained in Section 2.3.1, we have tried to limit these factors in various ways, such as putting the phone in airplane mode (but with working Wifi), disabling the battery charging, and killing processes that could have interfered with our measurements. Last, but not least, we have followed a consolidated practice in electric measurements, i.e., repeating the measures at least 30 times.

Threats to *external validity* concern the generalization of our findings. This study is admittedly limited to the energy consumption of Android APIs, whereas there could be other APIs that strongly contribute to the energy consumption of apps. Also, since the study has been conducted on a specific device, it could be the case that some APIs could consume more (or less) on other smartphones/tablets. Last, but not least, although we have selected a pretty varied set of 55 apps, we cannot claim they fully represent the universe of Android apps.

5. RELATED WORK

There are two threads of related work that we are interested to discuss. Firstly, we summarize existing approaches for tracing energy consumption in mobile apps. We also outline their pros and cons as well as similarities with the approach presented in this paper. Secondly, we explain the major differences between our study and previous work on finding energy problems in mobile phone apps. Also, while there are several existing approaches for mining API usage patterns [3, 24, 30, 41, 47], the goal of this work is rather on studying energy consumption of API patterns and not improving the state-of-the-art in API usage pattern mining. Thus, due to space limitations, we are not discussing all the related approaches and studies in detail, but rather summarize them across the most pertinent dimensions.

5.1 Energy Profiling in Mobile Phone Apps

While energy profiling is a broad research area encompassing architectures, operating systems, networking, and software engineering fields, in this section we position approach used in this paper in the context of the related work among the other techniques for energy profiling of mobile apps. In particular, we consider the following dimensions while classifying the related work (see Table 5): Apps - number of applications used in the evaluation of the proposed technique; Approach - approach used for collecting and estimating energy measurements, i.e., Hardware-based profiling (HBP), Power models (PM), Android Battery API (ABA); Profiling granularity, i.e., Application (A), Browser Activities (BA), Device Components (DC), Process (P), Flow Path (FP), Function (F), API Calls (APIC) and Statement (S); 4) Platform - underlying programming platform, i.e., Android (A) and Windows (W).

While our study is not the only one relying on hardware-based profiling approach, which is the most precise technique for measuring power consumption in mobile phones, it is the largest study up to date, surpassing existing studies by one order of magnitude in terms of the apps traced and analyzed. Moreover, our study is the only study that ensures statistical significance of the results since we executed major scenarios for each of the 55 applications at least 30 times. In fact, our study is the only study in the literature that uses HBP and does profiling of APIs calls. Pathak *et al.* [33] provide a first examination on the energy consumed by apps methods and the Android API. In this sense [33] could be considered as the closer paper to ours. The approach used in [33] also allows estimating power consumption at API level granularity and relies upon power models (PM). However, the study in [33] is not focused on API patterns and only a small set of Android API methods are reported as energy-greedy. Moreover, in our study we analyzed more apps and tested the scenarios 30 times to reduce the impact of race conditions on the measurements. Our approach also was able

Table 5: Related techniques on energy profiling.

Technique	Apps	Approach	Element	Platform
Flinn and Satyanarayanan[11]	1	HBP	P,F	A
Zhang *et al.* [46]	6	PM	A	A
Carroll and Heiser [6]	6	HBP	CD	A
Chung *et al.* [7]	4	HBP	F	A
Pathak *et al.* [34]	15	PM	DC	A,W
Pathak *et al.* [33]	22	PM	F,APIC	A,W
Thiagarajan *et al.* [40]	N/A	HWP	BA	A
Kapetanakis and Panagiotakis [23]	1	ABA,HBP	BA	A
Hao *et al.* [19]	5	PM	A,F	A
Xu *et al.* [43]	6	PM	DC	A
Hao *et al.*[20]	6	PM	A,FP,F,S	A
Li *et al.* [25]	5	HBP	S	A
Our work	**55**	**HBP**	**APIC**	**A**

Table 6: Related studies on energy bugs.

Study	Apps	Element	Platform	E-bugs	Prof.
Pathak *et al.* [32]	N/A	N/A	A,N	General	N/A
Zhang *et al.* [45]	15	O	A	Network	No
Pathak *et al.* [35]	86	FP	A	No-sleep	No
Vekris *et al.* [42]	328	APIC	A	No-sleep	No
Liu *et al.* [27]	6	BI	A	Sensors	No
Zang *et al.* [44]	6	APIC	A,iOS,W	No-sleep	No
Our work	**55**	**APIC**	**A**	**General**	**H**

to identify energy-greedy APIs that were not analyzed before (e.g., `SQLiteDatabase.query`, `Bitmap.getPixel(int,int)`).

5.2 Energy Bugs in Mobile Apps

There are several recent studies that aim at detecting a subset of energy bugs in mobile apps. We consider the following dimensions while classifying this related work (see Table 6): Apps - number of applications used in the study; Code - code elements representing the energy bugs, i.e., Objects(O), Flow paths (FP), API calls (APIC), and bytecode instructions (BI); Platform - underlying programming platform and operating system, i.e., Android (A), iOS, and Windows (W); E-bugs - types of energy bugs analyzed, i.e., Networking, sensor, no-sleep related, and general (or all) the possible types; Profiling - an indication whether any applications have been actually traced in the study (Exec) or only forums and/or repositories have been analyzed (Docs).

As it can be seen in Table 5, our study is the only pioneering study that focuses on finding energy bugs by actually power profiling the apps and analyzing API calls and patterns in the code of those apps. Other studies analyzed only forums and the repositories, but did not involve actual energy data collection on real apps. Also, our study aims at finding all possible types of energy problems (mostly focusing on those that were not previously identified in the literature), not only on those specifically related to networking or wakelock APIs, among the others.

6. CONCLUSION AND FUTURE WORK

This paper reports a study aimed at quantitatively and qualitatively investigating energy-greedy API calls and patterns identified from 55 free Android apps. To this aim, we have exercised 55 apps in the context of real usage scenarios, measured the energy consumption using a hardware meter, aligned such measurements with execution traces, and finally identified and traced onto source code the interesting API calls and patterns.

The obtained results allowed us to distill some pieces of actionable knowledge. Specifically, our findings indicate that some consolidated design and implementation practices, such as the use of Model-View Controller, information hiding, or else the implementation of the persistence layer through a relational database may have a non-negligible impact on the app energy consumption. For this reason, we suggested to developers to carefully ponder such choices, possibly pursuing alternative solutions, and balancing the tradeoff between a good design (i.e., high maintainability) and a better energy-aware solution.

Our work-in-progress goes towards different possible directions. First, for the sake of a better generalization of the obtained results we will sample and analyze further applications, possibly discovering patterns (and new categories) not encountered yet. Second, with the aim of providing further useful suggestions to mobile apps developers, we will compare the energy consumption of code implementing the same feature in similar apps, in order to identify the options that are more energy-efficient. Last, but not least, all the collected body of evidence can be used to build a catalogue of energy-aware patterns and anti-patterns in the context of mobile app development.

Acknowledgements

We thank Dr. Qun Li and his student Hao Han from the College of William and Mary for their help and advice with the power measurement process. We also thank Bogdan Dit from the College of William and Mary, Simone Scalabrino from the University of Molise, and Santiago Vargas-Baldrich from the National University of Colombia for their help in analyzing the data and verifying the results. This work is supported in part by the NSF CCF-1016868 and NSF CAREER CCF-1253837 grants. Gabriele Bavota and Massimiliano Di Penta are partially supported by the Markos project, funded by the European Commission under Contract Number FP7-317743. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

7. REFERENCES

[1] Android Monkey Recorder. `http://code.google.com/p/android-monkeyrunner-enhanced/`.

[2] How expensive findViewById ??. `https://groups.google.com/forum/?fromgroups=#!topic/android-developers/_22Z90dshoM`.

[3] M. Acharya, T. Xie, J. Pei, , and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *ESEC/FSE'07*, pages 25–34, 2007.

[4] I. Buchmann. How to define battery life.

[5] I. Buchmann. *Batteries in a Portable World.* Cadex Electronics Inc., 2011.

[6] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *USENIX ATC'10*, 2010.

[7] Y.-F. Chung, Ch-Y.Lin, and C.-T. King. Aneprof: Energy profiling for android java virtual machine and applications. In *ICPADS'11*, pages 372–379, 2011.

[8] W. J. Conover. *Practical Nonparametric Statistics.* Wiley, 3rd edition edition, 1998.

[9] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage determining the provenance of software development artifacts. *Empirical Software Engineering*, 2012.

[10] J. Davies, D. M. German, M. W. Godfrey, and A. J. Hindle. Software bertillonage: Finding the provenance of an entity. In *MSR'11*, 2011.

[11] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA'99*, pages 1–9, 1999.

[12] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine de Gruyter, New York, NY, 1967.

[13] Google. Android performance tips. `http://developer.android.com/training/articles/perf-tips.html`.

[14] Google. Android storage options. `http://developer.android.com/guide/topics/data/data-storage.html`.

[15] Google. Determining and monitoring the connectivity status.`http://developer.android.com/training/monitoring-device-state/connectivity-monitoring.html`.

[16] Google. Power profiles for android. availabel at `https://source.android.com/devices/tech/power.html`.

[17] Google. Profiling with traceview and dmtracedump. `http://developer.android.com/tools/debugging/debugging-tracing.html`.

[18] Google. Nexus 4 specifications. `http://www.google.com/nexus/4/specs/`, 2013.

[19] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Android applications' CPU energy usage via Bytecode profiling. In *GREENS'12*, pages 1–7, 2012.

[20] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE'13*, pages 92–101, 2013.

[21] A. Hindle. Green mining: A methodology of relating software change to power consumption. In *MSR'12*, pages 78–87, 2012.

[22] A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: Understanding and treating sleep conflicts in smartphones. In *EuroSys'13*, pages 253–266, 2013.

[23] K. Kapetanakis and S. Panagiotakis. Efficient energy consumption's measurement on Android devices. In *PCI'12*, pages 351–356, 2012.

[24] D. Kawrykow and M. P. Robillard. Detecting inefficient api usage. In *ICSE'09*, pages 183–186, 2009.

[25] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *ISSTA'13*, pages 78–89, 2013.

[26] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Online appendix of: Mining energy-greedy API usage patterns in Android apps: an empirical study. `www.cs.wm.edu/semeru/data/MSR14-android-energy`.

[27] Y. Liu, C. Xu, and S. C. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *PerCom'13*, pages 2–10, 2013.

[28] I. Mojica Ruiz, M. Nagappan, B. Adams, and A. Hassan. Understanding reuse in the Android market. In *ICPC'12*, pages 113–122, 2012.

[29] Moonsoon-Solutions. Power monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[30] H. A. Nguyen, T. T. Nguyen, W. Gary, A. T. Nguyen, K. Miryung, and T. N. Nguyen. A graph-based approach to API usage adaptation. In *OOPSLA'10*, pages 302–312, 2010.

[31] J. Ossher, H. Sajnani, and C. V. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *ICSM'11*, pages 283–292, 2011.

[32] A. Pathak, Y. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Hotnets'11*, page Article No 5, 2011.

[33] A. Pathak, Y. Hu, and M. Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *EuroSys'12*, pages 29–42, 2012.

[34] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y. M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Sixth Conference on Computer Systems (EuroSys'11)*, pages 153–168, 2011.

[35] A. Pathak, A. Jindal, Y. Hu, and S. P. Midkiff. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys'12*, pages 267–280, 2012.

[36] R Core Team. *R: A Language and Environment for Statistical Computing.* 2012. ISBN 3-900051-07-0.

[37] Stackoverflow. How often to call notifydatasetchanged() when changing arrayadapter. `http://stackoverflow.com/questions/15990849/`.

[38] Stackoverflow. Improving speed of getpixel() and setpixel() on Android bitmap. `http://stackoverflow.com/questions/4715840/`.

[39] Stackoverflow. Webview threads never stop. `http://stackoverflow.com/questions/2040963/`.

[40] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who killed my battery: Analyzing mobile browser energy consumption. In *WWW'12*, pages 41–50, 2012.

[41] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal API usage patterns. short paper. In *ASE'11*, pages 456–459, 2011.

[42] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal. Towards verifying Android apps for the absence of no-sleep energy bugs. In *HotPower'12*, 2012.

[43] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *NSDI'13*, pages 43–56, 2013.

[44] J. Zang, A. Musa, and W. Le. A comparison of energy bugs for smartphone platforms. In *MOBS'13*, 2013.

[45] L. Zhang, M. S. Gordon, R. P. Dick, Z. Morley, P. Dinda, and L. Yang. Adel: An automatic detector of energy leaks for smartphone applications. In *CODES+ISSS'12*, pages 363–372, 2012.

[46] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. Morley, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISSS'10*, pages 105–114, 2010.

[47] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009*, pages 318–343. Springer, 2009.