Detecting Energy Bugs and Hotspots in Mobile Apps

Abhijeet Banerjee[1] Lee Kee Chong[1] Sudipta Chattopadhyay[2] Abhik Roychoudhury[1]
[1]National University of Singapore, Singapore [2]Linköping University, Sweden
{abhijeet,cleekee}@comp.nus.edu.sg, sudipta.chattopadhyay@liu.se, abhik@comp.nus.edu.sg

ABSTRACT

Over the recent years, the popularity of smartphones has increased dramatically. This has lead to a widespread availability of smartphone applications. Since smartphones operate on a limited amount of battery power, it is important to develop tools and techniques that aid in energy-efficient application development. Energy inefficiencies in smartphone applications can broadly be categorized into *energy hotspots* and *energy bugs*. An *energy hotspot* can be described as a scenario where executing an application causes the smartphone to consume abnormally high amount of battery power, even though the utilization of its hardware resources is low. In contrast, an *energy bug* can be described as a scenario where a malfunctioning application prevents the smartphone from becoming idle, even after it has completed execution and there is no user activity.

In this paper, we present an *automated test generation framework* that detects energy hotspots/bugs in Android applications. Our framework systematically generates test inputs that are likely to capture energy hotspots/bugs. Each test input captures a sequence of user interactions (*e.g.* touches or taps on the smartphone screen) that leads to an energy hotspot/bug in the application. Evaluation with 30 freely-available Android applications from Google Play/F-Droid shows the efficacy of our framework in finding hotspots/bugs. Manual validation of the experimental results shows that our framework reports reasonably low number of false positives. Finally, we show the usage of the generated results by improving the energy-efficiency of some Android applications.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging, C.3 [Special-Purpose and Application-Based Systems]

Keywords: Mobile Apps; Non-Functional Testing; Energy Consumption

1. INTRODUCTION

Global penetration of smartphones has increased from 5% to 22% over the last five years. As of 2014, more than 1.4 billion smartphones are being used worldwide [1]. Over the recent years, smartphones have improved exponentially in terms of processing speed and memory capacity. This improvement has allowed application developers to create increasingly complex applications for such devices. Additionally, modern smartphones are equipped with a wide range of sensors and I/O components, such as GPS, WiFi, camera, and so on. These I/O components allow developers to create a diverse set of applications. In spite of such high computation power and developer flexibility, the usage of smartphones has been severely impeded by their limited battery capacity. In terms of computation capacity, most of the current-generation smartphones are two or even three orders of magnitudes better than their decade-old counterparts. However, the battery-life of these modern smartphones has improved only two or three times[1]. *High computational power coupled with small battery capacity and the application development in an energy-oblivious fashion can only lead to one situation: short battery life and an unsatisfied user base.*

Energy inefficiencies in smartphone applications can broadly be categorized into *energy hotspots* and *energy bugs*. An *energy hotspot* can be described as a scenario where executing an application causes the smartphone to consume abnormally high amount of battery power even though the utilization of its hardware resources is low. In contrast, an *energy bug* can be described as a scenario where a malfunctioning application prevents the smartphone from becoming *idle even after it has completed execution and there is no user activity*. Table 1 lists the different types of *energy bugs* and *energy hotspots* that can be found in Android applications. It is also worthwhile to know that most contemporary smartphone devices are designed to operate at different power states and prolong the battery life. However, as listed in Table 1, malfunctioning applications may lead to inappropriate power states, such as energy hungry GPS/sensor updates, *non-idle* power state in the absence of user activity and so on. Moreover, most of these energy inefficiencies appear when the application does not access the device resources in an appropriate fashion (*e.g.* not releasing WiFi/GPS/Wakelocks or expensive sensor updates), eventually hampering the battery life. Therefore, to build energy-efficient applications, it is crucial for the developer to know these energy inefficiencies in the application code. Presence of such energy inefficiencies in the application code can be highlighted to the developer via our proposed methodology.

In this paper, we present an automated test generation framework to detect energy hotspots/bugs in Android applications. Specifically, our framework systematically generates test inputs which are likely to capture energy hotspots/bugs. Each test case in our generated test suite captures a user interaction scenario that leads to an energy hotspot/bug in the respective application. We argue that

[1]For instance, if we compare `Nokia 9000 Communicator` (released in 1996) to `Samsung S3` (released in 2012), we can observe that the processing power has increased from `24MHz` to `1.4GHz`, whereas the battery capacity has only increased from `800mAH` to `2100mAH`

FSE '14, November 16–22, 2014, Hong Kong, China

#	Category	Energy Bug	Energy Hotspot
a	Hardware Resources	*Resource Leak*: Resources (such as the WiFi) that are acquired by an application during execution must be released before exiting or else they continue to be in a high-power state [2]	*Suboptimal Resource Binding*: Binding resources too early or releasing them too late causes them to be in high-power state longer than required [3], [4]
b	Sleep-state transition heuristics	*Wakelock Bug*: Wakelock is a power management mechanism in Android through which applications can indicate that the device needs to stay awake. However, improper usage of Wakelocks can cause the device to be stuck in a high-power state even after the application has finished execution. This situation is referred to as a Wakelock bug [5]	*Tail-Energy Hotspot*: Network components tend to linger in a high power state for a short-period of time after the workload imposed on them has completed. The energy consumed by the component between the period of time when the workload is finished and the component switches to the sleep-state is referred to as Tail Energy [6]. Note that tail energy does not contribute to any useful work by the component. Scattered usage of network components throughout the application code increases power loss due to Tail-Energy
c	Background Services	*Vacuous Background Services*: In the scenario where an application initiates a service such as location updates or sensor updates but doesn't removes the service explicitly before exiting, the service keep on reporting data even though no application needs it [7]	*Expensive Background Services*: Background services such as sensor updates can be configured to operate at different sampling rates. Unnecessarily high sampling rate may cause energy hotspots and therefore should be avoided. [8] Similarly, fine-grained location updates based on GPS are usually very power intensive and can be replaced by inexpensive, WiFi-based coarse-grained location updates, if an application is using both the WiFi and the GPS [9]
d	Defective Functionality	*Immortality Bug*: Buggy applications may respawn when they have been closed by the user, thereby continuing to consume energy [10]	*Loop-Energy Hotspot*: Portions of application code are repeatedly executed in a loop. For instance, a loop containing network login code may be executed repeatedly due to reasons such as unreachable server [10]

Table 1: Classification of Energy Bugs and Energy Hotspots

the systematic generation of such user interaction scenarios is challenging. This is primarily due to the absence of any extra-functional property (*e.g.* energy consumption) annotations in the application code. As a result, any naive test-generation strategy may either be *infeasible* in practice (*e.g.* exhaustive testing) or it may lead to an extremely poor coverage of the potential energy hotspots/bugs. This also brings us to the difficulty of defining an appropriate coverage metric for any test generation framework that aims to uncover energy hotspots/bugs. In our framework, we address these challenges by developing a directed search strategy for test generation.

To design a directed search strategy, it is critically important to know the potential sources of undesirable energy consumption. Table 1 lists such sources of energy consumption in Android applications. Moreover, existing works such as [11] have shown that I/O components are primary sources of energy consumption in a smartphone. One crucial observation is that I/O components are usually accessed in application code via *system calls*. Besides, the power management functionality (*e.g.* Wakelocks), background services and other hardware resources (*cf.* Table 1) of a device can only be accessed through a set of system calls. In summary, most of the classified energy hotspots/bugs (*cf.* Table 1) are exposed via the invocation of system call(s). Therefore, the general intuition behind our directed search strategy is to systematically generate user interaction scenarios which potentially invoke such system calls.

Our search strategy revolves around systematically traversing an event flow graph (EFG) [12]. EFG is an abstraction to capture a set of possible user interaction sequences. Each node in an EFG captures a specific user interaction (*e.g.* touching a button on smartphone screen), whereas an edge in the EFG captures a possible transition between two user interactions. Therefore, each trace in an EFG captures a possible sequence of user interactions. Since exhaustive enumeration of EFG traces is potentially infeasible, our directed search methodology generates appropriate EFG traces which are likely to lead to undesirable energy consumption. To accomplish this, we primarily employ two strategies. Based on our observation from Table 1, we execute selected EFG traces and these selected EFG traces invoke system calls that might be responsible for irregular power consumption . Besides, if an energy hotspot/bug is detected after executing an EFG trace, we record the sequence of system calls responsible for such irregular energy behaviour. Subsequently, we prioritize unexplored EFG traces that may invoke a similar sequence of system calls. Such a guidance heuristic primarily aims to uncover as many energy hotspots/bugs as possible in a limited time budget.

Besides the challenges encountered in generating energy stressing test inputs, it is also *non-trivial* to *automatically* detect a potential energy hotspot/bug in a given trace. To detect energy hotspots/ bugs, our framework executes a test input (*i.e.* a user interaction scenario) on a off-the-shelf smartphone, while simultaneously measuring the power consumption via a power meter. To detect an energy bug in a specific trace, we measure the statistical dissimilarities in power-consumption trace of the device, specifically, before and after executing the respective application. As the power consumption of an idle device should be similar, a statistical dissimilarity indicates an *energy bug*. To detect an energy hotspot, we employ an anomaly detection technique [13] to locate anomalous power consumption patterns. Once we finish the process of detecting hotspots/ bugs in a power-consumption trace, we generate a different user interaction scenario (using the directed search strategy in the EFG) to investigate. The test generation process continues till the time budget permits or all event traces invoking system calls have been explored. As the system calls are the potential locations to cause irregular energy behaviour, the quality of our test suite is provided via the coverage of system calls in the application.

Contribution. In summary, we provide a systematic definition, detection and exploration of energy hotspots/bugs in smartphone applications. We combine a graph-based search algorithm and guidance heuristics to find possible energy hotspots/bugs in an application. Each test case in our generated report captures a user interaction scenario that leads to an energy hotspot/bug. We have implemented our entire framework for Android apps. For the evaluation of our framework, we have performed experiments with 30 freely-available Android apps from the Google Play/F-Droid. These applications are diverse in terms of *apk* (Android application package file) size. The largest tested application was 8.0MB in size while the smallest application was 22KB in size. The average *apk* size of the tested applications was 1.1MB. The lines of code for the open-source applications used in our experiments varied from 448 to $11,612$, with an average of 4010 lines of code per application. Additionally, the applications used in our experiments use multiple I/O components and had a substantial number of user-interaction (UI) screens. Experiments with our framework uncovered energy bugs in 10 of the tested applications and energy hotspots in 3 of the tested applications. Manual validation of the experimental results shows that our framework reports reasonably low number of false positives. Finally, we show the usage of our test suite by improving the energy-efficiency of some of the tested Android applications.

2. GENERAL BACKGROUND

Android is an open-source operating system (OS) designed for mobile devices such as smartphones. We choose Android as our target platform primarily due to its relevance in the real world (globally 57% of all smartphones/tablets are Android based [14]). Additionally, a wide variety of tools are publicly available for Android application developers. This includes, among others, tools to monitor the state of an application in real-time (*e.g. logcat*), to communicate with the device (*e.g. android debug bridge*) and to facilitate application development and testing (*e.g. emulator*).

The user interaction interface of an Android application is referred to as an *Activity*. Figure 1 shows the life-cycle of an Android activity. An activity can be in one of the seven stages during its life-cycle. Usually, all the set-up tasks (such as acquiring resources and starting background services) take place in four stages of the activity, namely *onCreate*, *onStart*, *onResume* and *onRestart*. Similarly, all the tear-down tasks (such as releasing resources and stopping background services) take place in three stages, namely *onPause*, *onStop* and *onDestroy*. However, some real-life applications do not follow the ideal set-up and tear-down scenarios as explained via Figure 1. Such applications may contain *energy bugs*. This situation is made worse by the fact that most real-life applications have a huge number of feasible user interaction scenarios (due to complex GUIs). As a result, it can be impossible for a developer to test an application for all possible scenarios.

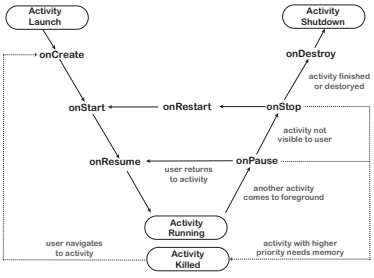


Figure 1: Life-cycle of an Android activity

Figure 2 shows a snippet of application code that has a potential energy bug. The application code is supposed to start a location-update background service (Line 10) in the *onCreate* method. Subsequently, it performs some operation with list data (Line 12). When the user stops the application, the location-update service is removed (Line 19) in the *onStop* method. However, if there is an exception before Line 19 (for instance, due to Line 18), the location

```
1  List<Integer> data;
2  LocationManager locationManager;
3  long Min_Update_Time = 10, Min_Distance = 1000 * 60 * 1;
4
5  @Override
6  public void onCreate(Bundle savedInstanceState) {
7    super.onCreate(savedInstanceState);
8    setContentView(R.layout.main);
9    locationManager = (LocationManager)getSystemService(LOCATION_SERVICE);
10   locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
11                                  Min_Update_Time, Min_Distance, this);
12   someFunctionToManipulateDataList();
13 }
14 @Override
15 public void onStop() {
16   super.onStop();
17   try {
18         data.clear(); //this can throw an exception
19         locationManager.removeUpdates(this);
20   } catch (Exception ex) {
21         Log.v("Demo", "some exception happended");
22   }
23 }
```

Figure 2: Code with a potential energy bug

update service is never stopped, resulting in an *energy bug*. The example in Figure 2 shows one possible scenario (*cf.* Table 1 #c: *Vacuous Background Services*) which can lead to an energy bug. Next, we shall show an example that can lead to an *energy hotspot*.

The code snippet in Figure 3(a) shows an example with energy hotspots due to disaggregated network activities (*cf.* Table 1 #b: *Tail-Energy Hotspot*). Observe that in Figure 3(a), network related code (Line 6) is interleaved with CPU-intensive code (Line 8) within the same loop. Such an interleaving causes energy-inefficiencies due to *Tail-Energy* (see Table 1 #b: *Tail-Energy Hotspot*). *Tail-Energy* behaviour has been observed for network components such as *3G*, *GSM* and *WiFi* [6]. Other works [11] have observed *Tail-Energy* in components such as storage disks and *GPS* as well. In order to reduce energy-loss due to *Tail-Energy*, the network related code in Figure 3(a) can be aggregated as shown in Figure 3(b).

```
1  public Object[] nonAggregatedComm()
2  {
3    Object[] objectArray =
4                    new Object[10];
5    for(int i=0; i<10; i++){
6      Object temp = downloadObject(i);
7      objectArray[i] =
8          processObject(temp);
9    }
10   return objectArray;
11 }
12
13
```

(a)

```
public Object[] aggregatedComm()
{
  Object[] tempArray = new Object[10];
  for(int i=0; i<10; i++){
    tempArray[i] = downloadObject(i);
  }
  Object[] objectArray = new Object[10];
  for(int i=0; i<10; i++){
    objectArray[i] =
        processObject(tempArray[i]);
  }
  return objectArray;
}
```

(b)

Figure 3: (a) Code with energy hotspot due to disaggregated communication (b) Code without energy hotspot

Finally, we shall explain the method used for obtaining the power consumption ratings of the hardware components in our smartphone. One approach to obtain the power consumption ratings would be to perform empirical experiments based on the guidelines provided on the Android developer web page [15]. However, there is a more elegant way to obtain the power consumption ratings. Most Android smartphones are shipped with a XML file (usually named as *power_profile.xml*) containing the average power consumption ratings for the hardware components in the device. The data contained in this XML file is provided by the device manufacturer and therefore it is reliable. Moreover, the Android framework uses this data to show battery related statistics. However, note that the data in this XML file is an indicator of average power consumption of the hardware components of the device and does not correspond to any particular application being run on the device. The data from *power_profile.xml* for our smartphone `LG L3 E400`, is shown in Figure 4.

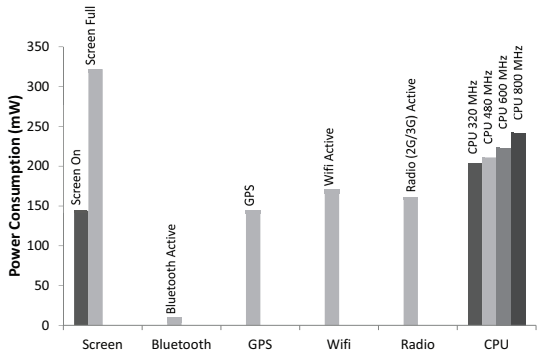


Figure 4: Power profile for `LG Optimus L3 E400` smartphone

3. FRAMEWORK OVERVIEW

An overview of our test-generation framework is shown in Figure 5. Our framework has two essential components: *(i)* guided exploration of selected event traces that are more likely to uncover energy hotspots/bugs, and *(ii)* detection of hotspots/bugs in a given event trace for an application. The information provided by the hotspot/bug detection component is also utilized by the guidance component to select subsequent event-traces. The process of selection, execution and detection continues until the given time-budget has expired or all event-traces invoking system-calls have been explored. Finally, event traces that lead to energy hotspots/bugs are reported to the developer for further investigation.

To detect a hotspot/bug, we measure the power consumption of the application for a given event-trace. However, it is impossible to detect a hotspot/bug in an application solely by analyzing its power consumption trace. For instance, consider a scenario where two programs P_1 and P_2 have similar power consumption traces. However, program P_1 has a much higher utilization of system resources (such as CPU) compared to P_2. In such a scenario, program P_1 is more energy-efficient than program P_2. Therefore, to accurately detect energy inefficiencies, it is important to define an appropriate metric for system-resource utilization.

For a hardware component x, the $Load_x$ represents the average amount of computational work performed by the hardware component x over a given period of time. $Load_x$ has a range from 0 to 1. For example, $Load_{CPU}$ represents the fraction of time CPU is in use and therefore $Load_{CPU}$ can be a number between 0 and 1. For other hardware components (*i.e.* WiFi, screen, Radio and GPS), $Load_x$ captures whether the respective components are in use. For instance, $Load_{WiFi}$ is set to 1 if the WiFi is transmitting data and it is set to 0 otherwise. For any hardware component x, we measure $Load_x$ directly from the device, while the application under test is being executed. It is important to note that a higher $Load_x$ in a high-power consuming component x would result in a higher power consumption for the device. Based on this information we define a new metric of *utilization* that will be subsequently used in energy hotspots/bugs detection.

DEFINITION 3.1. *Utilization (U) can be defined as the weighted sum of utilization rates of all major power consuming hardware components in a device, over a given period of time.*

Based on the power profile for our device (*cf.* Figure 4), major power consuming components in our mobile device are the screen, WiFi, Radio, GPS and CPU. Therefore, for a given time interval, the utilization of system resources can be computed by Equation 1.

$$\text{Utilization} = U_{Screen} + U_{CPU} + U_{WiFi} + U_{Radio} + U_{GPS} \quad (1)$$

$$U_{CPU} = \begin{cases} W_{CPU_{320}} \cdot Load_{CPU}, & \text{if CPU is operating at 320MHz} \\ W_{CPU_{480}} \cdot Load_{CPU}, & \text{if CPU is operating at 480MHz} \\ W_{CPU_{600}} \cdot Load_{CPU}, & \text{if CPU is operating at 600MHz} \\ W_{CPU_{800}} \cdot Load_{CPU}, & \text{if CPU is operating at 800MHz} \end{cases}$$

$$U_{Screen} = \begin{cases} W_{Screen_{ON}} \cdot Load_{screen}, & \text{if screen on} \\ W_{Screen_{FULL}} \cdot Load_{screen}, & \text{if at full brightness} \end{cases}$$

$$U_x = W_x \cdot Load_x, \quad x \in \{WiFi, Radio, GPS\}$$

In Equation 1, U_x represents the utilization of hardware component x. Utilization of a component x is directly proportional to its $Load_x$. For any component x, the value of W_x is computed from the power profile (Figure 4). Specifically, the value of W_x is normalized such that W_x for the most power consuming component is 1 (in our case $Screen_{FULL}$ as shown in Figure 4). Note that in our case Equation 1 does not include Bluetooth. This is because in our target device Bluetooth has a very low power consumption compared to other components. However, if required, we can easily extend Equation 1 to accommodate Bluetooth as well. Using the new metric of utilization (U), we can now compute the magnitude of energy-inefficiency as follows.

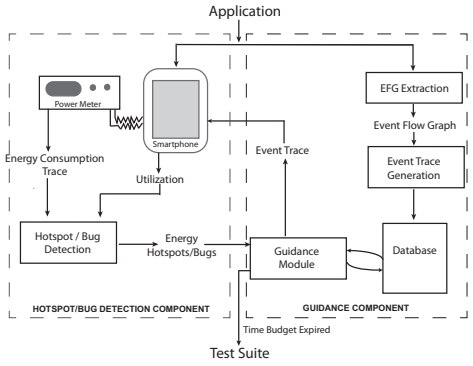


Figure 5: Overview of the test generation framework

DEFINITION 3.2. *Energy-consumption to Utilization (E/U) ratio is the measure of energy-inefficiency of an application for a given time period.*

If E/U ratio of an application is high, it implies that the energy-consumption is high, while utilization is low. Therefore, a high E/U ratio indicates that the application is *energy-inefficient*. Recall that we discuss two categories of energy issues that can make an application energy-inefficient (*i.e.* energy hotspots and energy bugs). A high E/U ratio during the execution of an application indicates the presence of an energy hotspot. On the contrary, a persistently high E/U ratio even after the application has completed execution indicates the presence an energy bug.

Now we shall briefly discuss the exploration of event traces to reveal hotspots/bugs. In our framework, guided exploration of selected event traces is based on event flow graph (EFG) [12]. EFG of an application can be defined as follows.

DEFINITION 3.3. *An Event Flow Graph (EFG) is a directed graph, capturing all possible user event sequences that might be executed via the graphical user interface (GUI). Nodes of an EFG represent GUI events. A directed edge between two EFG nodes X and Y represents that GUI event Y follows GUI event X.*

In our experiments, we use a modified version of the Dynodroid tool [16] to generate the EFG. Subsequently, our framework generates event sequences up to maximum length k and stores them in a database. After the event traces have been generated, our framework initiates a guided exploration of those traces. The crucial factor during the exploration is to identify the event traces that may lead to hotspots or bugs. Our framework accomplishes this by selecting event traces based on the number of invoked system calls and guidance heuristic. The guidance heuristic gathers information from previously detected hotspots/bugs, specifically the sequence of system calls which are likely to lead to energy inefficiencies. Subsequently, the selection process is biased towards event traces invoking a similar sequence of such system calls. This process of selection, execution and detection continues until the time-budget has expired or all event-traces invoking system-calls have been explored. Finally, event traces that lead to energy hotspots/bugs are reported to the developer for further investigation.

4. DETAILED METHODOLOGY

In the following sections, we shall describe our test generation methodology in detail. Broadly, our framework contains two substeps; *(i)* preprocessing the application under test to build a database of possible event traces, and *(ii)* test generation using event traces generated in the first step.

4.1 Preprocessing the application

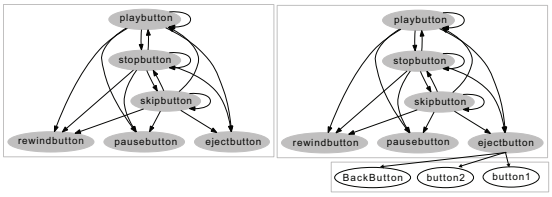
Preprocessing of application can be divided into three steps: (i) EFG extraction (ii) Event trace generation (iii) Extraction of system calls sequence for each event trace. Note that this preprocessing step is performed only once for each application. The generated EFG and database are stored for later use and need to be updated *only if* the application's user interface changes. Since this preprocessing is done *offline*, a developer can rerun the test generation step (detailed in Section 4.2) without repeating preprocessing step.

(i) Event Flow Graph extraction : We build the Event Flow Graph (EFG) based on the UI model proposed in [12]. For the purpose of EFG construction we use two third-party tools Hierarchy Viewer [17] and Dynodroid [16]. Hierarchy Viewer provides information about the UI elements of the application under execution and Dynodroid is used to explore these event sequence automatically. Note that Dynodroid does not generate the EFG by itself, therefore we modified the Dynodroid source code to build the EFG. The EFG was constructed gradually each time Dynodroid interacts with the application. Figure 6 shows how our EFG is being gradually built as Dynodroid performs the exploration of event sequences. It is worthwhile to note that Dynodroid does not guarantee to reach *all* GUI states during exploration. Therefore, our constructed EFG is in fact a *partial* EFG of the entire application. However, in our experiments, we observed that the generated EFGs cover most of the GUI elements for the tested applications.

(ii) Event trace generation : EFG is primarily used to generate a set of event traces. Note that each application has a start GUI screen. This GUI screen is presented to the user when an application is launched. We refer to this GUI screen as the *root screen*. Therefore, for a sequence of user interactions performed in an application, the first action corresponds to an event present in the *root screen*. Using this notion, we define an *event trace* as follows.

DEFINITION 4.1. *An event trace is defined as a path of arbitrary length in the EFG. Such a path must start from an event in the root screen of the respective application.*

Based on our EFG, we generate a complete set of event traces upto length k. These event traces are stored in a database for further analysis during test generation. Figure 6(b) shows the partial EFG of an application. The node containing the event $playbutton$ captures the *root screen* of the same application. An example *event trace* of length 3 would be $playbutton \rightarrow stopbutton \rightarrow playbutton$ or $skipbutton \rightarrow ejectbutton \rightarrow BackButton$. Note that events $playbutton$ and $skipbutton$ correspond to different events in the *root screen* of the application.



(a) (b)

Figure 6: EFG generation process (a) An example EFG (b) EFG after pressing "ejectbutton"

(iii) Extraction of system calls : Existing literature [11] has shown that I/O components are one of the major sources of energy consumption in smartphones. On observing the power profile of our smartphone (see figure 4) we find this argument to be consistent. In general, for modern smartphones the major power consuming components are the screen, CPU, WiFi, Radio, GPS, SDCard, Camera and Audio hardware. We observed that these components (except for the CPU) can only be accessed via a set of system calls (APIs) provided by the Android SDK framework. Therefore, we create a pool of such systems calls. Table 2 shows a categorization of these systems calls based on their functionalities. Since our target device (`LG L3 E400`) uses Android 2.3 (Gingerbread), therefore we only consider system calls available in Android 2.3. It is worthwhile to note that such a pool is constructed *only once* and it needs to be updated *only if the Android SDK framework changes*.

Functionality	Number of APIs	Example
Power Management	3135	WakeLock.acquire()
Local Area Wireless Networks	2116	WifiLock.acquire()
Telecomm Networks	1691	SmsManager.sendTextMessage()
Haptic Feedback	783	Vibrator.vibrate()
GPS	146	LocationManager.requestLocationUpdates()
Audio/Video	94	Camera.startPreview()
Storage	66	DownloadManager.enqueue()
Others	25	SensorManager.getAltitude()

Table 2: Categorization of Android system calls

We instrument the application code locations which invoke any system calls from our constructed pool. This instrumented code runs in an emulator on our desktop PC. The sole intention of this instrumentation is to collect the system call traces during the execution of an *event trace*. We execute the instrumented code on the emulator and record the system calls invoked for each *event trace*. These system calls are annotated with the EFG node corresponding to the triggered event. Thus, for each event trace generated from the EFG, we can generate the respective *system call trace*. It is important to note that the *event traces* are executed on the smartphone, as well as in the emulator. *The instrumented application runs on the emulator whereas the instrumentation-free application run on smartphone.* Therefore, the instrumentation does not influence the energy consumption behaviour of the application.

4.2 Test generation

In this subsection, we shall describe (i) technique for hotspot/bug detection (ii) guidance heuristic for the framework and (iii) algorithm for test-generation

(i) ***Technique of hotspot/bug detection***: As described in section 3, energy hotspots/bugs are those regions of code that lead to high E/U ratio (*cf.* Def 3.2). To detect energy hotspots during an event trace T, we must first obtain the E/U ratio trace (E/U_T), during the execution of T. E/U_T is divided into four different stages: pre-execution stage (PRE), execution stage (EXC), recovery stage (REC) and post execution stage ($POST$) (see Figure 7). The rationale for dividing E/U_T trace into four stages is as follows: in the PRE stage the execution of event trace T has not started yet. Therefore, PRE stage records the idle-behaviour (low-power state) of the device. Similarly, in the $POST$ stage, the devices has completed execution of T and so in an ideal scenario the device would have gone back to its idle-behaviour during $POST$ stage. The execution stage, as the name suggests, is when T is actually executing

on the device. After the execution of T, the device takes a brief period of time (referred to as *screen-time-out* duration) to return to its idle-behaviour. In our framework this time period between the EXC and $POST$ stage is referred to as REC stage [2].

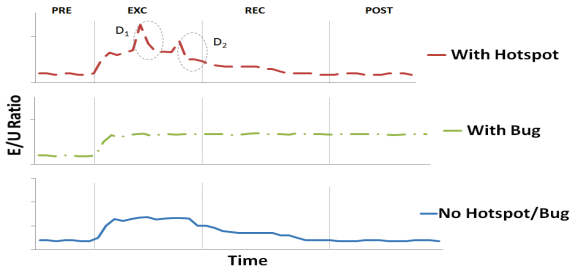


Figure 7: An example of energy-consumption to utilization (E/U) trace with no hotspot/bug, with an energy bug and with an energy hotspot

To detect the presence (or absence) of an energy bug we compare the E/U_T values in PRE and $POST$ stages using statistical methods. If the dissimilarity between E/U_T values in PRE and $POST$ stage is more than a predefined threshold (in our experiments the threshold was set to 50%), an energy bug is flagged (*i.e* execution of T changed the idle-behaviour of the device).

Compared to detection of bugs, detection of hotspots is much trickier. Hotspots may appear only during the execution of an event trace (*i.e.* EXC stage) or just after the execution of an event trace (*i.e.* REC stage) stage. Note that E/U_T values obtained in EXC stage and REC stage may substantially vary for different event traces. Besides, different executions of the same event trace may show different E/U_T values in EXC stage or REC stage, due to different hardware states. Therefore, we first need a clear definition of energy hotspots to detect them automatically. We believe that *abnormally high energy wastage during the execution of an event trace* is a suitable indicator of energy hotspots. To detect such unusual energy behaviours, we draw connections from the data mining and classification techniques. We observe that the problem of detecting unusual energy behaviours is similar to detect *unusual subsequences in time-series data*. We use an anomaly detection technique that computes *discords* [18] in a time-series data. *Discords* are subsequences in a time-series data, that are maximally different from the rest of the time-series. We employ the *discord* detector on the E/U_T values from the EXC and REC stage. As a result, the discord detector highlights subsequences in E/U_T that are abnormally different from the rest of the subsequences in the EXC and REC stage. Additionally, the anomaly detector also points out the magnitude of each computed discord. For instance, in Figure 7, discord D_1 has a higher anomaly magnitude than the discord D_2. These magnitudes are extremely helpful. This is because the computed energy hotspots can be *ranked* based on their magnitude, before reporting to the developers. As the anomaly detector, we integrate `JMotif` [13] into our framework. `JMotif` is an off-the-shelf data mining library and it includes the implementation of finding discords in a time-series data, as proposed in [18].

(ii) ***Guidance heuristics for test generation***: The primary objective of the guidance heuristics is to select an unexplored event trace that has a substantial likelihood of leading to a hotspot or a bug. The guidance function uses three parameters to rank the unexplored event traces: (a) number of systems calls in the event trace (b) similarity to previously explored, hotspot/bug revealing event traces (c) starvation of event traces due to unexplored system calls. The rationale for using these parameters is explained subsequently.

We have described in an earlier section (4.1:(iii) Extraction of system calls) that the major power consuming component in smartphones can be accessed through a set of system calls. Therefore, the presence of system calls that activate (or deactivate) such hardware components can be used for guiding our test generation. At the beginning of test generation process, all event traces are ranked according to the number of such system calls they can invoke. In subsequent iterations, the guidance module becomes more intelligent by learning specific system call subsequences that are more likely to generate energy hotspots/bugs, which is where the guidance by similarity (or exploration history) comes into play. While selecting an unexplored event trace, the guidance heuristics compares an unexplored trace to all previously explored event traces that had uncovered an energy hotspot or a bug. Comparison between two event traces is performed in terms of the sequence of systems calls they can invoke. Note that such a comparison is perfectly *feasible*, as we extracted the system call trace for each event trace during the preprocessing stage. Similarity between two system call traces is compared using *Jaro Winkler Distance* algorithm [19]. Finally, our third parameter, guidance by starvation, aims to cover as many system calls as possible during exploration. Since the first two parameters are based on the number of system calls and the exploration history it is possible that the guidance heuristics may ignore several unexplored system calls. This leads to *starvation*, where a set of system calls will never be explored by the test generation process. Such starvation is undesirable, as unexplored system calls may potentially expose new energy hotspots/bugs. Therefore, to ensure a fair coverage of all the system calls invoked by an application, we add a guidance parameter to deal with the problem of starvation. Essentially, guidance by starvation ranks all unexplored event traces by the ratio of number of unexplored system calls in an event trace to the total number of system calls in all event traces.

(iii) ***Algorithm for test-generation***: The algorithm for our test-generation framework is shown using a flow chart (see Figure 8). The primary objective of our framework is to uncover as many energy hotspots/bugs as possible in an application, within a given time budget. Input to our framework is an Android application from which the database of the application's event traces is generated. Recall that generation of event traces from the EFG of an applications was explained in section 4.1. Our framework systematically executes the event traces from the database on the smartphone. Each execution is monitored for presence of hotspots/bugs. The exploration continues until the allocated time budget has expired. On completion, the framework reports a set of event traces, each of which leads to an energy hotspot/bug when executed on the device. The two most important components of our framework, that are *Guidance heuristics for test generation* and *Technique of hotspot/bug detection*, have been discussed in preceding paragraphs. There is however, one more component of the framework that must be explained. Notice that in the flow chart (Figure 8), the first block indicates *Refine Guidance Parameters,* α, β, γ. Essentially, this indicates the step in our framework where the reliance (or the weight) of the various guidance parameters are refined. Recall that our guidance heuristics is based on three parameters, guidance by number of system calls (corresponding weight would be α), guidance by exploration history (corresponding weight would be β) and guidance by starvation of system calls (corresponding weight would be γ). Assume that, for a given event trace E, guidance by number of system calls assigns a rank G_n, similarly guidance by exploration history assigns a rank G_h and guidance by

[2] In all our experiments, REC stage was much larger than the *screen-time-out* duration. This allowed the device to return back to its idle behaviour by the $POST$ stage after a bug-free event trace has completed execution.

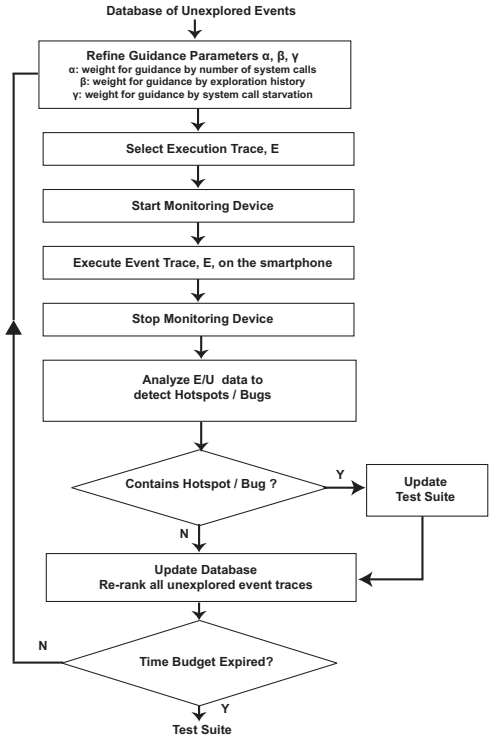


Figure 8: Flow chart for our test-generation framework

starvation assigns a rank G_s. To obtain a single score S_E for an unexplored event trace E, we use equation 2.

$$S_E = \alpha \times G_n + \beta \times G_h + \gamma \times G_s \quad (2)$$

where $\alpha + \beta + \gamma = 1$. In equation 2, α, β and γ are three tunable factors which drive the priorities of different guidance parameters. In the beginning, we do not have any knowledge about likely hotspots/bugs. Therefore, α is initialized to 1 and both β and γ are initialized to 0. In each iteration, the value of α, β and γ are refined to uncover likely energy hotspots/bugs, as well as to get a fair coverage of invoked system calls. Specifically, in each iteration, we decrease the value of α by a fixed amount Δ ($0 < \Delta < 1$). If an energy hotspot was found in the previous iteration, we increase the value of β to $\beta + \Delta$. The intuition behind this refinement is to find energy hotspots/bugs that had similar system call sub-sequences as previously found hotspots/bugs. We continue increasing the value of β as long as we find hotspots/bugs or the value of β reaches 1. If we are unable to find any hotspots/bugs in some iteration, we hope to reach previously unexplored system calls and therefore, we increase the weight of γ to $\gamma + \Delta$. This assignment of extra weight Δ is taken out from α, if $\alpha \geq \Delta$. Otherwise, we modify the value of β to $\beta - \Delta$ to decrease the priority of execution history.

5. EXPERIMENTAL EVALUATION

We evaluated our framework to answer the following three research questions: *(i)* Efficacy of our framework in uncovering energy bugs and hotspots in real-world applications, *(ii)* How can an application developer benefit from the reports generated by our framework, and *(iii)* Is guidance based on system call coverage more appropriate metric than code coverage for uncovering energy bugs and hotspots? First, we describe our experimental setup and the set of subject programs that we analysed in our experiments.

5.1 Experimental setup

In our experiments, we use an LG Optimus L3 smartphone as the device to run our subject programs. The device has a single core processor and features standard I/O components such as GPS, WiFi, 3G and Bluetooth. The device uses Android 2.3.3 (Gingerbread) operating system (OS). To monitor energy consumption of the smartphone, we used a Yokogawa WT210 [20] digital power meter for precise power measurement. Our energy-testing framework runs on top of a Desktop-pc that has an Intel Core i5 processor and 4 GB RAM. The OS used on our Desktop-pc was Windows 7.

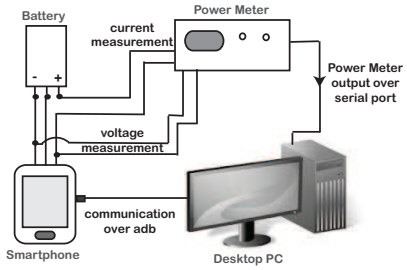


Figure 9: Our experimental setup

Figure 9 shows the setup for our experiments. For the purpose of this experiment we created a special apparatus to house the smartphone battery, such that we could measure the voltage and current flowing through the battery without any distortion. Note that contemporary smartphone batteries may have more than two terminals. Additional terminals may be used by the battery to report data such as internal temperature. However, for our experiments only the positive and the negative terminals need to be monitored (as shown in Figure 9). Any additional terminals may be directly connected to the smartphone. Our framework runs on the Desktop-PC, which also serves as the global clock. All the measurements from the power-meter (reporting power consumption data) and the smartphone (reporting utilization data) are collected at the Desktop-PC. Each reading is recorded with a timestamp generated on the Desktop-PC. Since the timestamps are generated by a single clock (the clock on the Desktop-PC) we can use these timestamps to synchronize [21] the data from the power-meter and the smartphone. Also note that we use the *android debug bridge* to communicate with the smartphone. These communication includes sending event traces to the smartphone and recording utilization data.

5.2 Choice of subject programs

The subject programs for our experiments are available on Google Play store/F-droid repository [22–51]. We have analyzed a total of 30 Android applications from different categories (*e.g.*tools, productivity, transportation) as shown in Figure 10. The subject programs are diverse in terms of *apk* (Android application package file) size. The largest application tested was 8.0MB in size while the smallest application was 22KB in size. The average *apk* size of the subject programs was 1.1MB. The subject programs also had varying GUI complexity. We measure GUI complexity of an application by the number of feasible event traces that could be explored, starting from the main screen of the application. By fixing the length of the event traces to explore (to a length of 4), we observe that our chosen subject programs contain between 26 to 2,800 feasible event traces. We also estimate the popularity of an application by observing the number of times it has been downloaded, as well as its user ratings. These two statistics are only available for applications on the Google Play store. As of March 10, 2014, the subject programs have an average user rating of 4.0 out 5, with

Application	Description	Feasible Traces (k=4)	Bugs Found / False Positive	Hotspots Found / False Positive	Hotspot / Bug Type	Previously Reported
Aagtl	A geocaching tool	131	Yes / No	Yes / Yes	Resource Leak	No
Aripuca	Records tracks and waypoints	502	Yes / No	No / n/a	Vacuous Background Services	No
Montreal Transit	Fetches bus, subway and other transit information	64	Yes / No	No / n/a	Expensive Background Services, Suboptimal resources binding	No
Omnidroid	Automated event/action manager	233	Yes / No	No / n/a	Vacuous Background Services, Immortality bug	Yes
Zamnim	Shows location-aware zmanim	965	Yes / No	No / n/a	Vacuous Background Services	Yes
Sensor Test	Monitors and logs sensor output	2,800	Yes / No	No / n/a	Immortality bug	No
Eponte	Displays traffic information	200	No / n/a	Yes / No	Suboptimal resources binding	No
760 KFMB AM	Listens to online radio	26	Yes / No	Yes / No	Vacuous Background Services, Suboptimal resources binding	No
Food Court	Finds restaurants near a location	42	Yes / No	No / n/a	Vacuous Background Services	No
Fire and Blood	Simple touch and draw game	156	Yes / No	No / n/a	Vacuous Background Services	No
Speedometer	Shows measurements of sensors	2,492	Yes / No	No / n/a	Vacuous Background Services	No

Table 3: Statistics for all the Energy Hotspots/Bugs found in tested applications (out of the 30 applications that we analyzed)

a minimum rating of 2.7 and a maximum rating of 4.6. The median number of downloads for the subject programs is between 10,000 - 50,000, with a minimum download count of 1,000 and a maximum download count of 10,000,000.

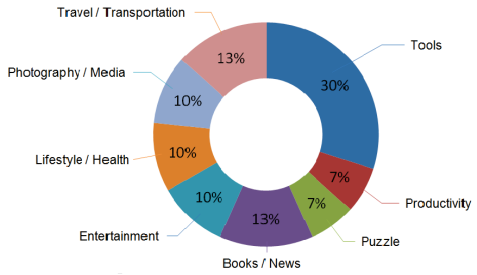


Figure 10: Categories of the 30 Android applications used in our experiments

Note that our framework does not require the source code of an application to detect energy hotspots/bugs. However, the source code is required to obtain code coverage metrics and for debugging purposes. Therefore, we only consider open-source applications for the second and third research questions (where the source code is needed to perform our evaluation). The lines of code for the open-source applications used in our experiments varied from 448 to 11,612, with an average of 4010 lines of code per application.

5.3 Results

RQ1: Efficacy of our framework in uncovering energy bugs and hotspots in real-world applications One of the objective of our experiments was to observe the efficacy of our framework to quickly uncover energy bugs and energy hotspots in real world applications. To do so, we evaluated the applications using our framework with a time budget of 20 minutes. Additionally, we also limit our exploration for event traces up to a length of 4. A summary of the bugs and hotspots reported by our framework is listed in Table 3.

Our framework reported energy bugs for 10 out of 30 subject programs. The framework also reported energy hotspots for 3 subject programs. Note that our hotspot detection technique is based on an anomaly detection method [18]. Therefore, some of the reported hotspots may contain *false positives* due to the presence of noise in the measured data. Such noise may arise due to unpredictable behaviours such as network load. To confirm a reported energy hotspot, we manually execute the respective event trace on our device and we observe whether the same energy hotspot can be replicated. The result of the manual validation (*cf.* Table 3) revealed only one *false positive*, for the application *Aagtl*. It is important to note that the number of *feasible* event traces can be substantially large even for event traces having length 4 (as shown in Table 3). In spite of this large number of event traces, we observed that our framework can quickly gravitate the exploration process towards more energy-consuming event traces. Existing tools for Android application UI testing , such as Monkey, cannot uncover such high energy consuming event traces because they are designed to stress test the UI of the application by generating pseudo random stream of user events irrespective of the application's EFG or the system call usage.

RQ2: How can an application developer benefit from the reports generated by our framework? After analyzing an application, our framework generates a *test report*. This report serves as a guide to optimize energy consumption and to remove potential energy issues. The report contains a set of test cases, where each test case captures an energy issue reported by our framework. Each test case includes *(i)* a *MonkeyRunner* script for automatic execution of events that lead to the energy issue , *(ii)* energy trace pattern *(iii)* details of the energy issue (*e.g.* magnitude of energy hotspot) and *(iv)* the set of system calls invoked.

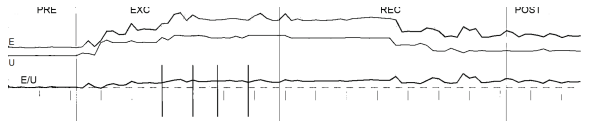
From the report, the developer may prioritize energy issues exhibiting an energy bug or an energy hotspot of relatively high magnitude. For each test case, the developer can run the provided *MonkeyRunner* script and observe the event sequence that navigates the application to trigger the reported energy issue. This would help the developer in identifying the root cause of the energy issue. For instance, let us assume that an event trace T exposes an energy hotspot. While executing T, if the hotspot appears before the execution of a certain event E, neither E nor any subsequent events in T are responsible for causing the hotspot. Thus, the search space for identifying the root cause of the hotspot is reduced to the code fragments that were executed before E was triggered. This will help the developer in fixing the reported energy issues. We have performed case studies on two of the analyzed applications (our framework reported energy bug for one and a hotspot for another) to demonstrate how a developer can utilize the generated reports to debug and fix energy issues in applications.

Aripuca GPS Tracker. Our framework reports two event traces with energy bugs in *Aripuca GPS Tracker*. The energy consumption pattern for such an event trace is shown in Figure 11(a). As shown in Figure 11(a), the energy consumption in the *POST* stage is not similar to the *PRE* stage, indicating an energy bug. Therefore, the device did not become *idle* even in the absence of user activity. The effect of the bug is permanent, unless *(i)* GPS location update is explicitly removed, or *(ii)* the application is killed. We

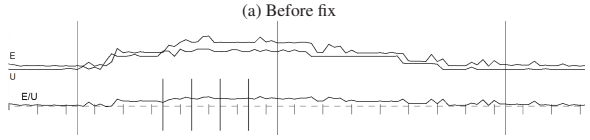
manually verified that the reported event traces do not exercise the functionality of the application that requires GPS location update to run in the background. The reported event traces were:

`waypointsButton` – `waypoint_details` – `MenuButton` – `button1`
`waypointsButton` – `waypoint_details` – `MenuButton` – `BackBtn`

The reported bug indicated that the location updates (GPS updates) were not removed before the application becomes inactive. By observing the similarity between the two traces (*i.e.* the event sequence `waypointsButton` – `waypoint_details` – `MenuButton`), we deduced that the bug was triggered upon arriving at a certain GUI state. We manually execute the event trace `waypointsButton` – `waypoint_details` – `MenuButton` and suspend the application afterwards by pressing the *Home* button. At this point, location updates are not needed by the application any more and they should be switched off. Upon inspecting the source code, we observed that the application had a missing code fragment for removing location updates when exiting. We fix the issue by adding the release code at an appropriate location. Thereafter, we re-run the reported event trace using our framework. The energy consumption graph after fixing is shown in Figure 11(b). As shown in Figure 11(b), the energy consumption in the *POST* stage is similar to the *PRE* stage, resolving the energy bug.



(a) Before fix

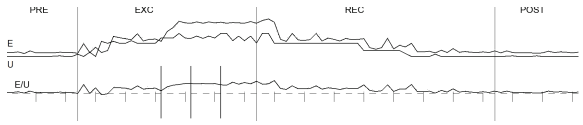


(b) After fix

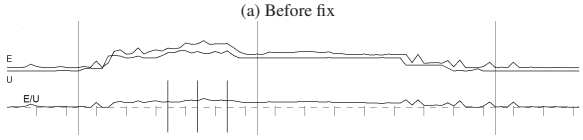
Figure 11: Energy trace of the event trace for *Aripuca GPS Tracker*

Montreal Transit. Five event traces with hotspots are reported for the application *Montreal Transit*. The energy consumption trace for one such event trace is shown in Figure 12(a). Immediately after the execution enters the *REC* stage (*cf.* Figure 12(a)), we can observe potentially high E/U ratios in a period of around 5 seconds. Note that this energy issue is an energy hotspot and not an energy bug. This is because the high E/U ratio does not persist. The code for pausing the application consumes abnormally high amount of energy, causing the hotspot to appear during the same period. We observed that all the five reported event traces exhibit similar hotspots. On a closer inspection, we found that the GPS location updates continue to run for a few seconds even after the application exits. Before we explain the exact cause for the hotspot, let us first give an overview of the application.

Montreal Transit is an application to show transit information, where each screen shows transit information for some mode of transportation. When a screen for some transportation, say subway, is displayed, it fetches the distances to some of the nearest subway stations. However, in order to do so, it needs to acquire the location of the device. Surprisingly, we found that the location update was triggered twice, instead of once. The second location update was triggered by a *third-party advertisement module* to display location-based advertisements in the application. We found that the code to load advertisement is being executed on the main thread of the application. As a result, any delay in loading the advertisement from the network prolongs the entire main thread. If the user exits the application while the main thread is being delayed, the release of GPS based location updates is delayed as well. The hotspots reported in our experiment can be best explained by such delay. To confirm our speculation, we moved the code related to the loading of advertisements in a separate asynchronous thread. As a result, we observed that the event traces which earlier exhibit hotspots, no longer do so (*cf.* Figure 12(b)). On a different note, we suggest that to develop energy-efficient applications, the developer should use expensive resources as optimally as possible. For instance, the location updates in the preceding scenario should be performed just once and shared between the various modules that need it. We also suggest that any feature that is surplus to the requirements of users (*e.g.* advertisements), should be put in a separate asynchronous thread to improve the user experience.



(a) Before fix



(b) After fix

Figure 12: Energy trace of the event trace for *Montreal Transit*

RQ3: Is guidance based on system call coverage more appropriate than code coverage for uncovering energy bugs and hotspots?

Application Name	System Call Coverage (%)	Code Coverage (%)	Lines of Code
Aagtl	100	21	11,612
Android Battery Dog	100	17	463
Aripuca	100	15	4,353
Kitchen Timer	100	30	1,101
Montreal Transit	89	11	10,925
NPR News	100	24	6,513
Omnidroid	83	36	6,130
Pedometer	100	56	849
Vanilla Music Player	86	20	4,081
Simple Chess Clock	100	49	448
WiFi ACE	100	27	504
World Clock	100	90	1,147

Table 4: Coverage statistics of all open-source applications used in our experiments

We have argued that I/O components and power management utilities contribute significantly to the energy consumption of a mobile device. Therefore, we use the number of system calls invoked by an event trace as one of the guiding parameters in the exploration. As a result, the test-suite generated by our framework should cover as many system calls as possible. On the other hand, a more conventional approach would be to measure code coverage of the test-suite to evaluate the efficacy of a test-generation framework. Therefore, we evaluated the efficacy of system call based coverage with respect to code coverage, to obtain a minimal test suite for uncovering energy bugs or energy hotspots. We choose *line of code* (LoC) as our code coverage metric and use *EMMA*, a Java code coverage tool, to obtain LoC covered by a test suite compared to the total LoC of the application. We observed that the generated test-suites had a system call coverage of more than 83%, while having code coverage ranging from $11\% - 90\%$ (see second and third columns of Table 4). Subsequently, we wanted to observe if achieving an incremental code coverage uncovers any additional hotspots/bugs. Therefore, we manually generated additional test cases for the applications in Table 4. We observed that the manu-

ally generated test cases increased the code coverage ranging from 4% to 17%. However, no additional hotspots/bugs were revealed. This is most likely due to the inefficiency of a human user to systematically find energy-inefficient event traces, based on a given metric. Additionally, on inspecting the *EMMA* coverage reports (and the source code), we observed that for real-life applications, a substantial portion of the code is present to give feedback to the user and to ensure compatibility over different versions of the OS. Therefore, the coverage achieved by executing such code would not necessarily contribute to finding energy hotspots/bugs.

6. RELATED WORK

In recent times, due to the prevalent use of smartphone devices, topics related to functional and extra-functional testing of smartphone applications have attracted the attention of software engineering research community. Recent proposals, such as [52] and [16], have discussed functionality testing of Android applications based on symbolic execution and biased random search. In contrast, we focus on automated testing of extra-functional aspects for smartphone applications, specifically the energy behaviour.

Recent works on energy-aware profiling [11, 53] have shown poor energy behaviour of several smartphone applications. These works on profiling validates the idea of energy-aware development for smartphone applications. However, like any other program profiling techniques, works proposed in [11, 53] require specific input scenarios to execute the application on smartphone device. A more recent work [54] has proposed a technique to relate power measurements with source lines of applications. Such a technique also requires input scenarios to execute an application. Automatically finding such input scenarios is extremely non-trivial, as the poor energy behaviour might be exposed only for a specific set of user interaction scenarios. Therefore, our approach on generation of input scenarios complements the works proposed on energy-aware profiling or source-line level energy estimation. Once the set of user interaction sequences is generated by our framework, they can be further used with works such as [11, 53] or [54].

The work proposed in [55] discusses energy-aware programming support via symbolic execution. For each code path explored by a symbolic execution toolkit, the base energy cost can be highlighted to the programmer. However, such an approach is preliminary in the sense that it only considers the CPU power consumption. In contrast, power consumption to access memory subsystems, network card and other I/O components were not considered. In smartphone devices, I/O components consume the most power. Since we perform direct power measurements for an application, we can highlight the gross energy consumption to the developer, without ignoring the energy consumption of any hardware component. The work in [56] proposes to analyze the overall energy behaviour of an application via an energy model. Our goal is orthogonal to such approach. We aim to find user interaction scenarios that may lead to undesirable energy behaviours of an application. Therefore, our work has a significant testing flavour compared to the work proposed in [56]. More importantly, we rely on direct power measurements rather than relying on any energy model. Another work [57] uses *data flow analysis* to detect *wakelock* bugs in Android applications. The detection of wakelock bugs is relatively easy. This is due to the fact that the acquire and release of wakelocks can be related directly to program statements. Therefore, the detection of wakelock bugs can be performed even in the absence of power measurements. In contrast, we aim to solve a more general problem of detecting energy inefficiencies and in addition, we also compute the specific input scenarios that witness the same.

A different line of work aims to produce energy-efficient applications from different implementations of the same functionality [58,59]. The decision to choose an implementation is influenced by monitoring the power consumption for a *given test-suite*. For instance, the work in [58] dynamically chooses approximate implementations of a given functionality to reduce the power consumption. Along the same line, a recent work [59] monitors the power consumption of different API implementations and computes the potentially best implementation in terms of energy-efficiency. Our work is complementary to such approaches, as we aim to automatically detect input scenarios that result in energy inefficiencies and generate a test-suite that can be used for improving the energy-efficiency of the application. Finally, the work in [60] introduces programming language constructs to annotate energy information in the source code. Since we directly measure the power consumption, our approach does not require any new language construct.

7. DISCUSSION AND FUTURE WORK

Summary. In this paper, we provide a systematic definition, detection and exploration of energy hotspots/bugs in smartphone applications. Our methodology is used to develop a test-generation framework that targets Android applications. Each entry in our generated test report contains a sequence of user-interactions that leads to a substantial wastage of battery power. Such test cases are useful to understand several corner scenarios in an application, in terms of energy consumption. Our evaluation with 30 applications from Google Play/F-Droid suggest that our framework can quickly uncover potential energy hotspots/bugs in real-life applications.

Threats to validity. It is worthwhile to mention that our test generation method is not *complete*. This is due to the fact that our computed event flow graph (EFG) may only cover a portion of the application. As a result, we may not expose *all the energy hotspots/bugs* in an application. Besides, our current test generation framework revolves around directing the test generation towards I/O operations, as I/O components are some of the major sources of energy consumption in smartphones. However, it is possible in some pathological cases (*e.g.* unusual cache thrashing and memory traffic) that CPU-bound applications may lead to substantial drainage of battery power. Detection of such energy stressing behaviours can be studied in the future.

In our current implementation, we can deal with GUI-based applications by generating UI inputs automatically. However, certain applications (such as *game applications*) require human intelligence in navigating through the different GUI screens. For example, the transition between two GUI screens might happen only by answering questions that require human intelligence. In such a situation, we may not be able to generate sufficient event traces automatically to stress the energy behaviour.

Future work. Using our energy-aware test generation framework, several research directions can be studied in future. In particular, we plan to study energy-aware debugging. Specifically, we plan to use our energy stressing input scenarios and compute the *root cause* of energy wastage *automatically*. Such debugging techniques will greatly help the developer to bring down the possible causes of energy wastage. We also plan to investigate the automated refactoring of an application to reduce energy wastage.

8. ACKNOWLEDGEMENT

The work was partially supported by a Singapore MoE Tier 2 grant MOE2013-T2-1-115 entitled "Energy aware programming".

9. REFERENCES

[1] Business insider: Smartphone and tablet penetration. http://www.businessinsider.com/smartphone-and-tablet-penetration-2013-10?IR=T.

[2] Android develeoper website, wifimanager. http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html.

[3] Android application coding guidelines -power save. http://dl-developer.sonymobile.com/documentation/dw-300012-Android_Power_Save.pdf.

[4] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter. Removing energy code smells with reengineering services. 2012.

[5] Android develeoper website, powermanager. http://developer.android.com/reference/android/os/PowerManager.html.

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

[7] Android develeoper website, sensormanager. http://developer.android.com/reference/android/hardware/SensorManager.html.

[8] Android-sensors. http://developer.android.com/guide/topics/sensors/sensors_overview.html.

[9] Android develeoper website, location strategies. http://developer.android.com/guide/topics/location/strategies.html.

[10] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*.

[11] 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.

[12] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, 2003.

[13] JMOTIF: a time series data-mining toolkit based on SAX and TFIDF statistics. http://code.google.com/p/jmotif/.

[14] Business insider: Number of smartphones worldwide. http://www.businessinsider.com/15-billion-smartphones-in-the-world-22013-2?IR=T.

[15] Android power profiles. http://source.android.com/devices/tech/power.html.

[16] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for android apps. In *ESEC/SIGSOFT FSE*, 2013.

[17] Hierarchy viewer. http://developer.android.com/tools/help/hierarchy-viewer.html.

[18] E. J. Keogh, J. Lin, and A. W-C. Fu. HOT SAX: Efficiently finding the most unusual time series subsequence. In *ICDM*, 2005.

[19] W. E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, 1990.

[20] Yokogawa wt210 digital power meter. http://tmi.yokogawa.com/us/products/digital-power-analyzers/power-measurement-application-software/wtviewer-for-wt210wt230/.

[21] A.S. Tanenbaum and M. van Steen. *Distributed systems: principles and paradigms*. Pearson Prentice Hall, 2007.

[22] Android advanced geocachingtool. https://play.google.com/store/apps/details?id=com.zoffcc.applications.aagtl.

[23] Android battery dog. https://play.google.com/store/apps/details?id=net.sf.andbatdog.batterydog.

[24] Aripuca gps tracker. https://play.google.com/store/apps/details?id=com.aripuca.tracker.

[25] Kitchen timer. https://play.google.com/store/apps/details?id=com.leinardi.kitchentimer.

[26] Montreal transit. https://play.google.com/store/apps/details?id=org.montrealtransit.android.

[27] Npr news. https://play.google.com/store/apps/details?id=org.npr.android.news.

[28] Pedometer. https://play.google.com/store/apps/details?id=name.bagi.levente.pedometer.

[29] Simple chess clock. https://play.google.com/store/apps/details?id=com.chessclock.android.

[30] Wifi advanced config editor. https://play.google.com/store/apps/details?id=org.marcus905.wifi.ace.

[31] World clock. https://play.google.com/store/apps/details?id=com.irahul.worldclock.

[32] Sensor status. https://play.google.com/store/apps/details?id=com.tpaln.snsst.

[33] Zoom camera. https://play.google.com/store/apps/details?id=ar.com.moula.zoomcamera.

[34] Voice recorder. https://play.google.com/store/apps/details?id=si.matejpikovnik.voice.pageindicator.

[35] Virtual recorder. https://play.google.com/store/apps/details?id=ix.com.android.VirtualRecorder.

[36] Quick recorder. https://play.google.com/store/apps/details?id=com.workspace.QuickRecorder.

[37] Speedometer. https://play.google.com/store/apps/details?id=com.bjcreative.tachometer.

[38] Zmanim. https://play.google.com/store/apps/details?id=com.gindin.zmanim.android.

[39] Omnidroid. https://play.google.com/store/apps/details?id=edu.nyu.cs.omnidroid.app.

[40] Fox news. https://play.google.com/store/apps/details?id=com.foxnews.android.

[41] Best unit converter. https://play.google.com/store/apps/details?id=simple.a.

[42] Sensor tester. https://play.google.com/store/apps/details?id=com.dicotomica.sensortester.

[43] Eponte. https://play.google.com/store/apps/details?id=com.amoralabs.eponte&hl=en.

[44] Goodreads. https://play.google.com/store/apps/details?id=com.goodreads.

[45] Food court. https://play.google.com/store/apps/details?id=com.eksavant.fc.ui.

[46] Fire and blood. https://play.google.com/store/apps/details?id=com.zeddev.plasma2.

[47] 760 kfmb am. https://play.google.com/store/apps/details?id=com.airkast.KFMBAM.

[48] Math workout. https://play.google.com/store/apps/details?id=com.akbur.mathsworkout.

[49] Vanilla music. https://play.google.com/store/apps/details?id=ch.blinkenlights.android.vanilla.

[50] Vimeo. https://play.google.com/store/apps/details?id=com.vimeo.android.videoapp.

[51] Baby solid food. https://play.google.com/store/apps/details?id=com.bytecontrol.diversification.

[52] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*, 2012.

[53] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *MobiSys*, 2011.

[54] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *ISSTA*, 2013.

[55] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat. SEEP: exploiting symbolic execution for energy-aware programming. HotPower, 2011.

[56] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, 2013.

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

[58] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[59] I. Manotas, L. Pollock, and J. Clause. SEEDS: A software engineer's energy-optimization decision support framework. In *ICSE*. ACM/IEEE, 2014.

[60] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *OOPSLA*, 2012.