# Debugging Energy-efficiency Related Field Failures in Mobile Apps

Abhijeet Banerjee
National Univ. of Singapore
abhijeet@comp.nus.edu.sg

Hai-Feng Guo
Univ. of Nebraska at Omaha, USA
haifengguo@unomaha.edu

Abhik Roychoudhury
National Univ. of Singapore
abhik@comp.nus.edu.sg

## ABSTRACT

Debugging *field failures* can be a challenging task for app-developers. Insufficient or unreliable information, improper assumptions and multitude of devices (smartphones) being used, are just some of the many factors that may contribute to its challenges. In this work, we design and develop an *open-source framework that helps to communicate, localize and patch energy consumption related field failures in Android apps*. Our framework consists of two sets of *automated tools*: one for the app-user to precisely record and report *field failures* observed in real-life apps, and the other assists the developer by automatically localizing the reported defects and suggesting patch locations. More specifically, the tools on the developer's side consist of an Eclipse-plugin that detects specific patterns of Android API calls, that are indicative of energy-inefficient behaviour. In our experiments with real-life apps we observed that our framework can localize defects in a short amount of time (~3 seconds), even for apps with thousands of lines-of-code. Additionally, the energy savings generated as a result of the patched defects are significant (observed energy savings of up to 29%). When comparing the patch locations suggested by our framework to the changes in the patched code from real-life app-repositories, we observed a significant correlation (changes suggested by our tool also appeared in the source-code commits where the reported defects were marked as fixed).

**Categories and Subject Descriptors:** D.2.5 [Software Engineering]: Testing and Debugging

**Keywords:** Mobile Apps; Debugging; Energy-efficiency

## 1. INTRODUCTION

Debugging *field failures*, even for functionality related defects, can be a challenging process for the developers [1–4]. It can be even more challenging in the case of non-functional defects (such as energy-inefficiencies) because such defects depend not only on the failure-revealing inputs but also on the state of the hardware device on which the app is executed. Online coding repositories often provide tools such as issue tracking systems, discussion forums, etc, to alleviate this problem, however, in many scenarios such tools are insufficient. Consider the example of Issue 520 (*Battery drain when not in use*) for the app *MyTracks* [5] (more than 10 Million downloads on Google Play Store). Since the original commenter had reported this issue, a total of 43 people (including project members) have participated in the discussion over a period of 4 years to debug the issue. Participants of the discussion have provided test-cases and device descriptions, and exchanged various versions of app files, snapshots, log files, changed devices, etc. However, these ad-hoc methods have done anything but to further confuse the involved parties. Here is an excerpt from their conversation.

| | |
|---|---|
| Comment 1<br>Jul 21, 2011 | When I don't use MyTracks, I don't expect to see it at the top of the list of applications draining the battery |
| Comment 3<br>Jul 23, 2011 | I don't really understand this issue. I checked the log but I did not see any suspicious. . . . |
| Comment 5<br>Jul 27, 2011 | Answer to comment 3: if you examine the log related to comment 2, you will see that MyTracks has never been launched but the application still won the third prize for power consumption. |
| Comment 35<br>Oct 3, 2012 | . . . Using a battery monitor called "GSam Battery Monitor", it give more information about what "My Tracks" is using: "Orientation", and it's the only things use by "My Tracks" when it does not track anything (no CPU, no wakelock ..) |
| Comment 40<br>Jul 13, 2013 | In response to comment 35, we have removed the "Orientation" sensor in My Tracks version 2.0.5 |
| Comment 41<br>Jul 13, 2013 | I need help to debug this further. . . . wondering . . . I can email you a few APKs to try. |

This example goes on to show how challenging it could be to communicate and debug defects in real-world apps even when both the parties (the user who reports the defect and the developer) were willing to cooperate. As highlighted in our previous work [6], a mobile app may demonstrate energy-inefficient behaviour due to a number of reasons (*cf* Table 2). Some of these defects may only manifest when complex sequence of interactions (or patterns) occur during the execution of the app. For example, suboptimal resource binding (binding resource too-early or unbinding too late) may make an app energy-inefficient. Another example could be the scattered usage of network components that cause power loss due to *Tail Energy* [7]. To localise such energy-consumption related defects, the developer needs to track relevant (energy/computation intensive) Android API calls in a context-sensitive manner (*where*, *when*, and *how* within the activity life-cycle). However, such contextual debugging information is much beyond user's capability to collect for communication. As a result, often the information exchanged between the two parties is insufficient, vague and sometimes even misleading.

*What is needed to solve this problem is a framework that can provide a reliable yet succinct means of communicating user-observed defects and an automated technique to use this communicated information to localize and debug these defects on the developer side.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*MobileSoft'16,* May 16-17, 2016, Austin, TX, USA
Copyright 2016 ACM 978-1-4503-4178-3/16/05 ...$15.00
http://dx.doi.org/10.1145/2897073.2897085 .

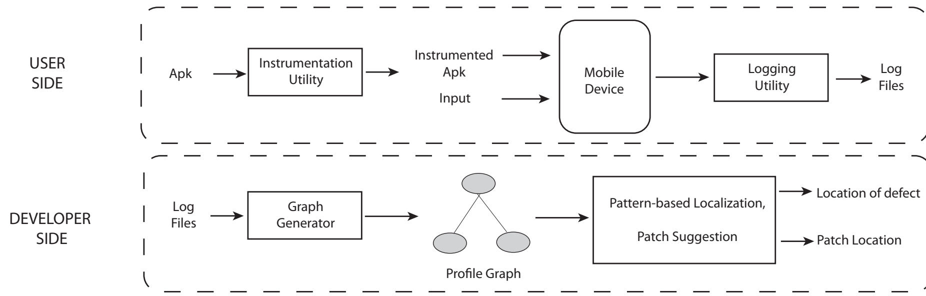



**Figure 1: Overview of log-based, energy-inefficiency localization in mobile apps**

In this paper we present a framework that can break the communication barriers between users and developers on assisting in energy-aware debugging. Figure 1 presents an overview of our framework, which contains two main parts: one for the user and the other for the developer. The user side tools consist of an instrumentation utility, which can automatically instrument a given apk file, followed by a customized logging utility, which records the log messages and system states at runtime. To report a defect the user simply executes the instrumented app, with the failure-revealing test inputs, on her mobile device, while the logging utility is running. The log file recorded automatically thus contains all the information that is needed to localize/patch the defect and can be sent to the developer.

The developer side tool consists of an Eclipse plugin, in which the debugging framework has been embedded. Provided with the log file, the debugging framework generates a profile call graph, which is subsequently analysed through a contextual analyzer to identify Android API call patterns for energy-inefficiencies. The pattern-based fault-localization technique is motivated by the fault-model presented in [6]. If such patterns are found, the framework localizes the defect in the app source-code and also suggests potential patch locations. It also provides a visual representation of observed defect. The framework on the developer side works in an automated fashion and requires minimal configuration.

**Key Contributions:**

- We present a practical framework for debugging energy consumption related *field failures* in mobile apps. Our framework assists the app-user to precisely report the observed defect, and assists the developer to localize, visualize and patch the reported defect.
- We present a set of Android API call patterns that are indicative of energy-inefficient behaviour in mobile apps. These patterns are motivated by the fault-model in [6]. We also provide real-life examples (user-reported issues) for each defect type.
- We present a context-sensitive analyzer that detects patterns of energy-inefficient behaviour within the profile call graph of the app, that is generated from the user-provided log-files capturing the faulty-scenario. The context, in particular, captures the dependence among the relevant (energy and/or computation intensive) APIs and the life-cycles stages of an activity.
- We provide an open-source implementation of our framework at `http://www.comp.nus.edu.sg/~rpembed/energydebugger/`. As far as we know, our toolset is the first that provides support for energy-aware debugging. Evaluation of our framework with real-life apps has shown that it can localize defects in short amount of time (~3 seconds), even for apps with thousands of lines-of-code. Additionally, the energy savings generated as a result of the patched defects are significant (observed energy savings up to 29%). When comparing the patch locations suggested by our framework to the changes in the patched code from real-life app-repositories, we observed a significant correlation (changes suggested by our framework also appeared in the source-code commits that indicated the observed energy-defect to be fixed).

# 2. DETAILED METHODOLOGY

In this section we shall discuss the (i) Instrumentation & Logging, (ii) Profile Graph Generation, (iii) Pattern-based contextual analysis for energy-inefficient behaviour detection and (iv) Defect localization and patch suggestion.

## 2.1 Instrumentation & Logging

Before the app can be executed to generate the log files, it has to be instrumented by our tool. Table 1 lists some of the packages from which methods are instrumented by our tool. These methods can be divided into two categories (i) event-handlers, that are invoked on arrival of events and (ii) Android API call that may have significant impact on the energy consumption. For example, the package `android.app.activity` (in the category event handlers), houses the method that are called when an activity is created (onCreate) or when an activity is paused (onPause), etc. Whereas, the package `android.hardware.*` (in the category Android API calls), houses the methods that are used to access hardware components such as `Sensors` and `Cameras`.

**Table 1: Event-handlers and Android API calls that are instrumented**

| Category | Type | Package Name |
|---|---|---|
| Event Handlers | Activity | `android.app.activity` |
| | Service | `android.app.service` |
| | Receiver | `android.content.BroadcastReceiver` |
| Android API calls | Bluetooth | `android.bluetooth.*` |
| | Sensors | `android.hardware.*` |
| | Location | `android.location.*` |
| | Multimedia | `android.media.*` |
| | PowerManager | `android.os.PowerManager.*` |
| | Database | `android.content.SQLite.*` |
| | SMS | `android.telephony.SmsManager` |
| | Telephony | `android.telephony.TelephonyManager` |
| | Network (A) | `android.net.*` |
| | Network (J) | `java.net.URL` |
| | IO | `java.io.*` |
| | Cipher | `javax.crypto.Cipher` |
| | Apache Http | `org.apache.http.*` |
| | Thread | `java.lang.thread` |

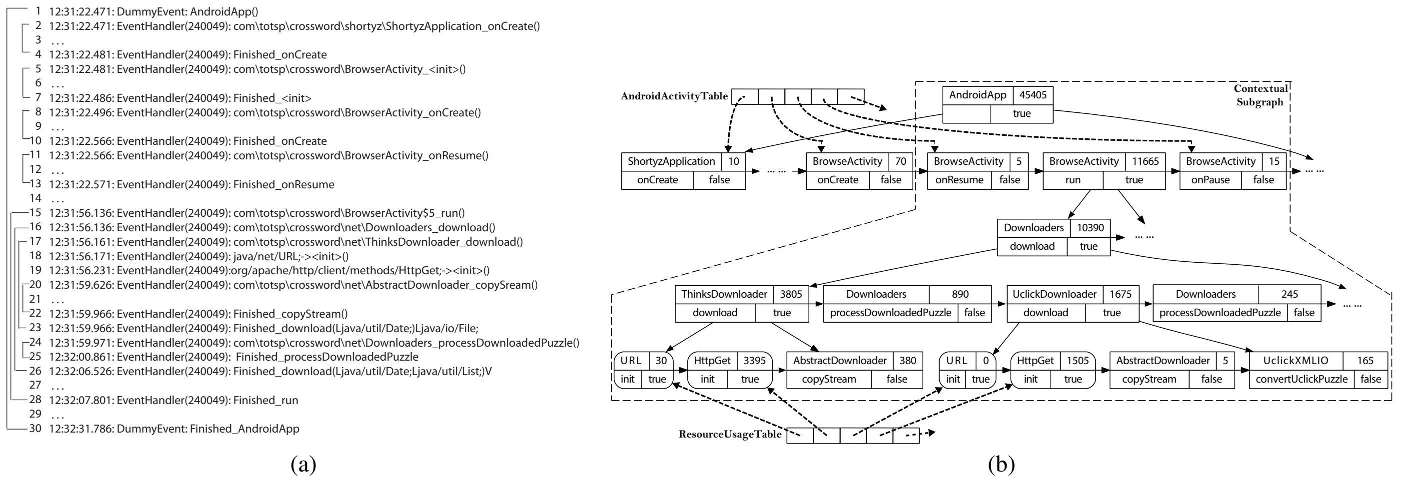



**Figure 2: (a) Log-messages generated from Shortyz app (b) Partial profile call graph for Shortyz app**

Figure 3 shows an example code after instrumentation. Observe that for the event-handler onCreate both `start` and `end` have been instrumented (lines 3 and 11, respectively). Additionally, the log messages contain the thread id which helps in resolving ambiguity in presence of multiple threads. All Android API calls which belong to packages described in Table 1 are instrumented in a similar manner (lines 5 and 6). For certain Android API call we need to add an additional line of instrumentation to record their parameters. This because for such Android API calls, their parameters may decide the resultant power consumption. For instance, in Figure 3, the parameters to the API call `setFlashMode` decide the state of flash (off, on, torch, etc) post invocation. Therefore, an additional line (*cf.* line 8) has been instrumented.

It is worthwhile to know the instrumentation tool works on apk file (source files not needed) and is very light weight, hence it can be run on any commodity machine on the user side. Also it is relatively easy to use as it does not require any prior configuration. Once the target apk files have been instrumented, the app-user (who wishes to report the defect) can execute the instrumented app on her mobile device. This execution should be done while our logging utility is monitoring over android debug bridge (adb). To do so, the user simply needs to connect the mobile-device through a USB cable to the PC on which the logging utility is running. The logging utility specifically monitors and records the log messages that are instrumented by the logging utility. In addition, it also records information relevant to device's power consumption, such as status of wakelocks, frequency of GPS updates, etc.

```
 1 public void onCreate(Bundle paramBundle)
 2 {
 3   Log.i("EventHandler","PackageName\ActivityName\onCreate"+Process.myTid());
 4   //...
 5   Log.v("AndroidAPI", "Landroid/hardware/Camera;->open()");
 6   Camera localCamera = Camera.open();
 7   //...
 8   Log.i("Parameters", "Landroid/hardware/Camera$Parameters;->setFlashMode" + "off");
 9   Log.v("AndroidAPI", "Landroid/hardware/Camera$Parameters;->setFlashMode");
10   parameters.setFlashMode("off");
11   Log.i("EventHandler", "Finished_onCreate");
12 }
```

**Figure 3: Example of code instrumentation**

## 2.2 Profile Graph Generation

The energy profile graph for an Android app is a dynamic call graph [8], that represents call dependencies between various subroutines that are executed in a given execution of the app. It primarily emphasizes on the method invocations related to energy consumption and app's GUI *activity* life-cycle. Call graph profiling [8–11] has been an effective method of program analysis and has been used in applications such as performance analysis and compiler optimizations. For example, in context of performance analysis, call-graph profiling has been used to monitor per-subroutine time consumption, based on which performance bottlenecks can be identified. The energy profile graph is based on similar principles but it is bit more sophisticated than the standard call-graph. Not only does it record the amount of time spent within each application component, but also records the behaviour of energy-intensive resources (such as the GPS, Screen, Wifi, etc) and status of activity life-cycle within a given application. It is worthwhile to note that such a call graph is possible because of instrumentation as described in section 2.1.

The log (generated using the logging utility, *cf.* Figure 1) is a well-structured, hierarchical text file. The entries in the log file are ordered by time-stamps. Figure 2(a) shows an example of such a log file. Note that since an Android app is event-driven, it does not have an explicit `main` method. Therefore, for the purpose of clarity we introduce a pair of dummy events, labelled `AndroidApp()` and `Finished_AndroidApp()`, that represents the start and end of execution of the app. Such a pair of entry-exit events forms the basis of the nested hierarchical structure for the call graph, since for each event-handler $e$, all of its subroutines, including method-invocations, are logged within the (entry-exit) pair of events for $e$. In Figure 2, such an entry-exit pair can be found on lines 15 and 28, respectively, and examples of method-invocation can be found on lines 18 and 19.

During the log-file parsing (to generate the profile call graph), we also create two axillary data structures: *AndroidActivityTable* and *ResourceActivityTable*. As the name suggests, the *AndroidActivityTable* keeps track of activity related nodes (such as `onCreate`, `onResume`) in the profile call graph, and *ResourceActivityTable* keeps track of nodes related to energy-intensive resources. The actual processing of the log-file happens in two passes:

*First Pass*: For each event-handler, entry-exit events are matched. Method-invocations that are related to energy-intensive resources are recorded in the *ResourceActivityTable*. Activity stages are recorded in *AndroidActivityTable*. Two implementation issues that may arise during this pass are as follows:

**Abnormal Exits** Abnormal exits (from method calls) may happen due to runtime exceptions. This may create unmatched method call (entry-exit) pairs in the log file. To detect such

issues, we maintain an environment stack. When an exit event does not match with the top entry event in the environment stack, it implies that the method call corresponding to the top entry event, may have had an abnormal exit.

**Multi-Threading** Log messages from multiple simultaneously executing threads (within the app) can also disrupt the balanced entry-exit pairs in the log file. To counter this issue, our runtime logging utility records the thread ID for each logged event. This allows us to do the entry-exit pair matching for each thread.

*Second Pass*: Well-balanced, (entry-exit) paired events are used to construct a hierarchical call-dependency graph, where each node contains the following fields supporting the first-child/next-sibling representation:

```
class Node {
  Node firstChild;    // first subroutine child
  Node nextSibling;   // next sibling subroutine
  Node parent;        // parent caller
  String className;   // class for current method
  String methodName;  // current method name
  long timeStamp;     // logging timeStamp
  long runtime;       // runtime between entry-exit events
  boolean MPCC;       // if method contains Android API calls
  ...                 // of interest
}
```

Figure 2(b) shows a partial call graph for one of the analysed apps, where rectangular nodes are used to represent event-handler nodes and round nodes are used to represent Android API calls.

## 2.3 Patterns for Energy-inefficient Behaviour

Table 2 presents a list of energy-inefficiency related defects commonly observed in mobile apps. For each defect category, we present the pattern of Android API calls that indicate energy-inefficient behaviour, the set of affected components and a real-world example along with user comments, where such a defect has been observed. These defect patterns are derived from the energy-inefficiency fault-model presented in our previous work [6].

Most of the patterns ($P_{defectcategory}$) in Table 2 are represented using a context-free grammar. The symbols $a_{component}, u_{component}$ and $r_{component}$ are used to denote the Android API calls for acquire, usage and release of a component. Whereas, the symbol $x_{\overline{component}}$ is used to denote all Android API calls *not* associated with a component. The symbols $\epsilon, a, u, r, x$ are used as terminals, whereas the symbols $Q$ and $X$ are used as variables. Rest of symbols used in the patterns retain their conventional meaning.

The patterns for $P_{\text{immortality bug}}$ and $P_{\text{loop hotspot}}$ require some additional explanation as they use non-standard notations. $P_{\text{immortality bug}}$, in particular, is a disjunction of three separate scenarios. Specifically, if patterns consistent with $P_{\text{resource leaks}}$, $P_{\text{wakelock bugs}}$ or $P_{\text{vacuous services}}$ are detected and localized to a program location which is restarted every time the system (mobile device) is rebooted, then our framework reports the presence of an immortality bug. For this defect category, we define two additional functions *PL* and *LaunchonReboot*. The function, *PL* : $P_{defectcategory} \rightarrow location$, provides the program location ($\langle class, method, lineno \rangle$) for a given pattern (specifically the beginning of the pattern). The function, *LaunchOnReboot* : $location \rightarrow boolean$, is used to determine the feasibility of a given program location to be launched at reboot. It is worthwhile to know that for Android apps, it is possible to design such a function because the relevant information (which methods will be launched on reboot) is provided statically by means of `AndroidManifest.xml`. Specifically, the function *LaunchOnReboot* keeps a map of all `Receivers` or `Activities` that have registered for the `BOOT_COMPLETED` broadcast intent.

Loop-energy hotspot ($P_{\text{loop hotspot}}$) represents the scenario where an energy hotspot, *i.e.* $P_{\text{suboptimal binding}}$, $P_{\text{tail energy}}$ or $P_{\text{expensive services}}$, is localized to a loop in the app source code. Being executed in a loop can magnify the impact of such defects and therefore may need additional re-factoring effort. An additional function, *InLoop* : $location \rightarrow boolean$, is added which can tell if a program location falls within the boundaries of a loop construct (*i.e.* `for, while, do-while`). In the following paragraphs, we shall outline the overall algorithm for pattern-detection. However, for the purposes of brevity we shall limit our discussion to only two defect patterns ($P_{\text{resource leaks}}$ and $P_{\text{tail energy}}$).

## 2.4 Contextual Analysis for Energy-inefficient Pattern Detection

The fault-localization process begins by analysing the contextual dependency among Android API calls, activity events, and other work loads, within the profile call graph. Our contextual analysis focuses on the innermost activity cycle for each energy-related Android API call to identify patterns that are associated with energy-inefficient behaviour. The main purpose of our contextual analysis is to extract a contextual subgraph which contains the following information:

**WHEN** the runtime scenario in which the foreground application activity lies, e.g., `Activity`.*onResume*() and `Activity`.*onPause*(). The *when*-context will help developers to replay the scenario when energy bugs or hotspots happen.

**HOW** the sequence of Android API call showing how the related hardware resource is acquired, used, and released, if any. The *how*-context is valuable to identifying the faulty patterns and categorizing the type of bugs or hotspots.

**WHERE** for each Android API call involved in *when* and *how*-context, the method and class details of Android API call, the caller details of Android API call (where it was actually invoked), as well as the subsequent calls to that Android API call are gathered. The subsequent calls are done to resolve ambiguity in case its caller invokes that Android API call multiple times. The *where*-context is critical to fault localization in source code.

**Algorithm 1** Energy Defect Detection

```
 1: Global: AT – an abbreviation for AndroidActivityTable
 2:         RT – an abbreviation for ResourceUsageTable
 3: Input: resourcesInUse – resources used in the app
 4: Output: a collection of energy defects, if any
 5: function DEFECTDETECTION(resourceInUse)
 6:     ds ← ∅
 7:     for (each h : resourceInUse) do
 8:         for (each defect : h.defectList) do
 9:             switch (defect) do
10:                 case "resource leaks":            ▷ Algorithm 2
11:                     ... ...
12:                     break
13:                 case "tail-energy hotspot":       ▷ Algorithm 3
14:                     ... ...
15:             end switch
16:         end for
17:     end for
18:     return ds
19: end function
```

**Table 2: List of energy-inefficiency related defects as presented in the work of [6]. For each defect category, we present the defect pattern, patch suggestion and affected hardware components and also provide a real-world example, along with user comments.**

| | Defect Type | Affected Components | Defect Pattern (P) | Patch Suggestion | Real-world example | |
|---|---|---|---|---|---|---|
| Hardware Resources | Resource Leaks | I/O Components<br>$i \in$ {Wifi, Sensor, GPS, Camera } | $P_{\text{resource leak}} \to a_i\ Q$<br>$Q \to X_{\bar{i}} \mid X_{\bar{i}}\ u_i\ Q$<br>$X_{\bar{i}} \to \epsilon \mid x_{\bar{i}}\ X_{\bar{i}}$ | Release $a_i$ after PL($P_{\text{resource leak}}$) | Csipsimple Issue 81 [12] | "When I run Sipdroid, by the end of the day I have battery indicator at about 50%. When I run CSipSimple instead, with all other activities basically the same, by end of the day the battery indicator is yellow or even red (< 20% I think)." |
| | Suboptimal Resource Binding | | $P_{\text{suboptimal binding}} \to a_i\ Q\ r_i$<br>$Q \to X_{\bar{i}}\ u_i\ X_{\bar{i}} \mid X_{\bar{i}}\ u_i\ Q$<br>$X_{\bar{i}} \to \epsilon \mid x_{\bar{i}}\ X_{\bar{i}}$ | Re-factor $X_{\bar{i}}$ | Sofia Public Transport Issue 38 [13] | "Disable GPS updating when estimates dialogue is displayed . . . This will " reduce battery usage |
| Sleep-state Transition Heuristics | Wakelock Bug | Power Management<br>$p \in$ {CPU, Screen, Keypad}<br><br>$c \in$ {CPU} | $P_{\text{wakelock bug}} \to a_p\ Q\ X_{\bar{p}}$<br>$Q \to X_{\bar{p}} \mid a_p\ Q\ r_p \mid Q\ Q$<br>$X_{\bar{p}} \to \epsilon \mid x_{\bar{p}}\ X_{\bar{p}}$ | Add $r_p$ to match the reference count of $a_p$ | Adw Launcher Android Issue 202 [14] | "This could lead to some battery drain, or with our AMOLED screens - burn in. I discovered this by accidentally hitting the app drawer button when setting my phone on my desk, I assumed it would turn it self off but after 15 minutes never did." |
| | Tail-energy Hotspot | Network<br>$n \in$ { Wifi, 4G, 3G,2G } | $P_{\text{tail energy}} \to a_n\ Q\ r_n$<br>$Q \to X_c\ u_n\ X_c \mid X_c\ u_n\ Q$<br>$X_c \to \epsilon \mid x_c\ X_c$ | Re-factor $X_c$ | Google Voice Location Issue 4 [15] | Feature Suggestion: Optimize Data Usage . . . Totally agreed. I've used this app on three different phones, and I've seen noticeable battery drainage when it's active." |
| Background Services | Vacuous Services | Background Services<br><br>$s \in$ {GPS, Sensors, Wifi,4G,3G,2G } | $P_{\text{vacuous services}} \to a_s\ Q$<br>$Q \to X_{\bar{s}} \mid X_{\bar{s}}\ u_s\ Q$<br>$X_{\bar{s}} \to \epsilon \mid x_{\bar{s}}\ X_{\bar{s}}$ | Stop service $a_s$ after PL($P_{\text{vacuous services}}$) | Recycle-locator Issue 33 [16] | "Battery life is absolutely terrible . . .It appears to be due to the GPS constantly searching for a signal." |
| | Expensive Services | | $P_{\text{expensive services}} \to a_s\ u_s\ r_s$<br>If configuration parameters of $a_s$ do not follow energy-efficiency guidelines [18] | Change configuration parameters for $a_s$ | Osmdroid Issue 76 [17] | "MyLocationOverlay class uses 0 as default values for requesting location updates. 0 (fastest updates possible) is maybe an overkill and drains some battery" . . . |
| Defective Functionality | Immortality Bug | All of the above | $P_{\text{immortality bug}} \equiv$ {<br>LaunchonReboot ( PL ($P_{\text{resource leak}}$))<br>$\vee$<br>LaunchonReboot( PL ($P_{\text{wakelock bug}}$))<br>$\vee$<br>LaunchonReboot( PL ($P_{\text{vacuous services}}$))<br>} | Use patch suggestion for respective issue. May re-factor PL($< pattern >$) from methods that launch on reboot | Omnidroid Issue 98 [19] | "Battery Drain Concerns . . . I know it was omnidroid because I looked at the battery use graph and omnidroid was using 8 times more battery than the next highest app. It dwarfed everything else." (app restarts itself on reboot) |
| | Loop-energy Hotspot | | $P_{\text{loop hotspot}} \equiv$ {<br>InLoop( PL ($P_{\text{tail energy}}$))<br>$\vee$<br>InLoop( PL ($P_{\text{suboptimal binding}}$))<br>$\vee$<br>InLoop( PL ($P_{\text{expensive services}}$))<br>} | Use patch suggestion for respective issue. May re-factor PL($< pattern >$) out of loops | K9Mail Issue 424 [20] | "I found today that when my mail accounts were temporarily unreachable due to a server glitch . . . battery drained *very* quickly while the phone became noticeably warmer even though it was in standby mode with the screen display off. Also, k9mail is AFAIK the only third-party background service that I have running" |

$a$ : Android API call to acquires a I/O component(h)/power management utility(p)[1]/services(s)[2]
$u$ : Android API call that uses a I/O component(h)/power management utility(p)/service(s)
$r$ : Android API call to release a I/O component(h)/power management utility(p)/service(s)
$x$ : any Android API call

$PL : P_{defectcategory} \to location$
$LaunchonReboot : location \to boolean$
$InLoop : location \to boolean$

1 : power management utilities can be reference counted
2 : of the many possible services, only the energy-intensive services, such as the once using sensors, are monitored

Algorithm 1 shows the skeleton of the main procedure, named DEFECTDETECTION, on how to detect all the relevant energy bugs or hotspots given a list of resources in use. For each resource $h$ in use, we prepare $h.defectList$, a list of possible bug or hotspot types associated with $h$. Our defect detection procedure will perform contextual analysis to investigate the resource usage in an energy profile subgraph to find out whether the runtime trace matches a defect pattern from a list of possible resource-specific defects. Note that AT and RT are abbreviations for AndroidActivityTable and ResourceUsageTable, respectively.

**Algorithm 2** Contextual Analysis for Resource Leak Bugs

```
1:  for (i ← 0 to RT.size() −1) do              ▷ each Android API call RT[i]
2:      if (RT[i] ∈ a_h) then                   ▷ Android API call to acquire h
3:          bTime ← FOREGROUNDBEGIN(AT, i)
4:          eTime ← FOREGROUNDEND(AT, i)
5:          j ← i + 1
6:          paired ← false
7:          while (not paired and j < RT.size() and
8:                  RT[j].timeStamp < eTime) do
9:              if (RT[j] ∈ r_h) then           ▷ Android API call to release h
10:                 paired ← true
11:             end if
12:             j ← j + 1
13:         end while
14:         if (not paired) then
15:             defect.type ← "resource leaks"   ▷ new a defect
16:             defect.info ← {RT[i]}
17:             ds ← ds ∪ {defect}
18:         end if
19:     end if
20: end for
```

*Detecting $P_{\text{resource leaks}}$* : Hardware components that are acquired by an app during its execution must be released before exiting, or else, the resources continue to be in high-power state and keep consuming energy. Typically, resources that are acquired during the setup stages of an activity (such as onResume()) should be released during its tear-down stages (such as onPause()). Failing to follow such protocols may lead to resource leaks. Such scenarios are represented using defect pattern $P_{\text{resource leaks}}$ in Table 2. Algorithm 2 outlines the analysis required to detect such a defect pattern within the contextual dependency subgraph. For each Android API call for resource acquisition (line 2), we first capture its contextual dependency subgraph by locating the foreground running activity while the resource $h$ has been acquired. To determine the temporal boundary of the contextual dependency subgraph, we use the procedures FOREGROUNDBEGIN and FOREGROUNDEND (lines 3-4). These two procedures can be implemented by searching the data-structure AndroidActivityTable for the innermost activity cycle just wrapping the Android API calls for resource acquisition. Subsequently, our framework checks for a release Android API call for the resource that has been acquired within the subgraph (7-13). Finally, if the release Android API call is not found a defect is recorded (line 14). The data-structure *defect.info* (line 16) maintains a sufficient set of method signatures related to *when*- and *how*-context so that the contextual subgraph can be easily retrieved for fault-localization and defect-visualization.

*Detecting $P_{\text{tail energy}}$*: It is common for network components in a mobile device to linger in a high power state, for a short-period of time, after the imposed workload has completed [7]. Such heuristics can reduce the time it takes to transit the device from a low-power (idle) state to an high power (active) state, when the subsequent (network) request arrives. However, while network component waits in a higher-power state, no transmission takes place but additional energy is consumed. The additional energy consumption is referred to as Tail Energy and can be minimized to increase the energy-efficiency of an app. The defect pattern ($P_{\text{tail energy}}$) from Table 2 represents such a scenario. Algorithm 3 shows the contextual analysis required to identify the presence of such a defect pattern. Similar to Algorithm 2, we begin by obtaining the contextual subgraph (lines 4-5). Subsequently, the pattern $P_{\text{tail energy}}$ is searched for in the subgraph (lines 7-14). More specifically, the procedure PROFILEGRAPHBROWSE(RT[$i$], RT[$j$]) traverses the profile graph from the node RT[$i$] to node RT[$j$], in a temporally-sequential order, (equivalent to pre-order traversal), while collecting CPU intensive processes ($X_c$) between resource usage Android API calls.

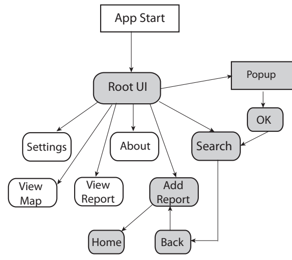
**Algorithm 3** Contextual Analysis for Tail-Energy Hotspot

```
1:  i ← 0
2:  while (i < RT.size()) do
3:      if (RT[i] ∈ u_h) then                   ▷ Android API call to use h
4:          bTime ← FOREGROUNDBEGIN(AT, i)
5:          eTime ← FOREGROUNDEND(AT, i)
6:          j ← i + 1              ▷ to find the next resource usage index j
7:          while (RT[j].timeStamp < eTime) do
8:              if (RT[j] ∈ u_h) then
9:                  defects.info ← defects.info ∪
10:                         PROFILEGRAPHBROWSE(RT[i],RT[j])
11:                 i ← j
12:             end if
13:             j ← j + 1
14:         end while
15:         if (defect is not empty) then
16:             defect.type ← "tail-energy"          ▷ new a defect
17:             ds ← ds ∪ {defect}                  ▷ add a new defect
18:         end if
19:         i ← j
20:     else
21:         i ← i + 1
22:     end if
23: end while
```

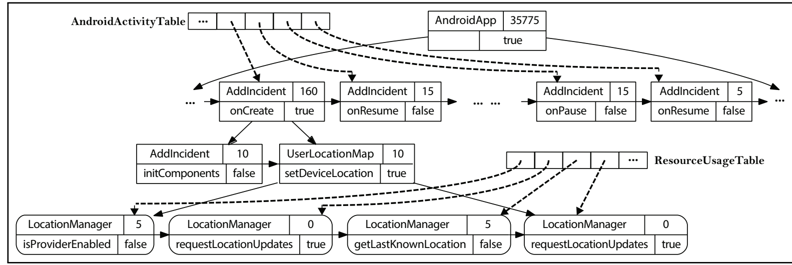
## 2.5 Defect Localization & Patch Suggestion

Finding the presence of energy-inefficient patterns is only a part of the debugging effort. The other part involves locating the defect, or rather the origin of defect, within the app-source code. The contextual analyzer on a profile call graph returns a set of detected defects, each of which contains a defect type and a sequence of evidential method signatures, ordered by time-stamps. The sequence of evidential methods, depending on the defect type as outlined in Table 2, may contain (a) *when*-context, the events related to foreground activity stages, (b) *how*-context, Android API calls where a hardware component is acquired, used, or released, or (c) other Android API calls not associated with the involving hardware component, but with noticeable computational time, e.g., greater than 5 milliseconds. Each evidential method $\alpha$ has detailed signatures which include $\alpha$'s prototype and its Class information, its caller's prototype and Class information, and the details of $\alpha$'s subsequent calls in its caller method. The subsequent calls are used to resolve the ambiguity in case that there are multiple occurrences of $\alpha$ in its caller method. We assume that the app-developer has access to the app source-code while debugging. To localize a defect, our framework simply highlights the defect type and displays the defect scenario by illustrating the sequence of evidential methods, and further allows users to select each of evidential method to reach the source code where it was invoked for understanding the reported defect.

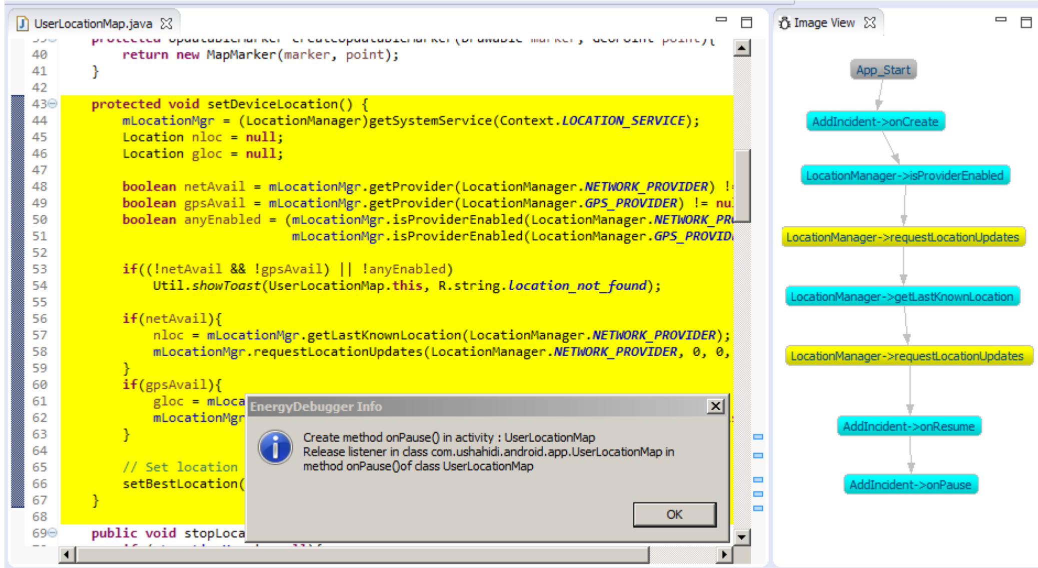
Our framework also gives concrete patch suggestions, as out-




(a) Some GUI Elements




(b) Contextual Profile Sub-graph




(c) Snapshot showing defect localization, visualization and patch suggestion

**Figure 4: Debugging *Ushaihdi Android***

lined in Table 2, for each reported defect. For example, on the defect of resource-leaks where the acquisition Android API call, $a_i$, misses a corresponding release statement, it is suggested that a release Android API call may be added in $onPause$. There may be multiple alternative locations (in the source code), where the patch may be added. Specifically, in the case of energy bugs, where the acquired services/resource/power management utilities can be released in several locations in the program. Our framework takes a conservative approach in suggesting patch-location for energy bugs, suggesting the $onPause$ event handler as the patch location for the activity that acquires the service/resource/utility, since for each activity that comes to the foreground, the $onPause$ event handler is called whenever it leaves the foreground. For another example, in the scenario of tail-energy hotspot, where usages of network components $u_n$ are scattered with noticeable irrelevant methods $X_c$ in-between, it is suggested that all $u_n$'s with a same network component should be grouped together, if possible, and followed by the irrelevant methods $X_c$'s. Similarly, for the loop-energy hotspot where tail-energy hotspot occurs within a loop structure, it is suggested that the irrelevant methods $X_c$'s should be handled in a separated subsequent loop, if possible, for energy saving.

## 3. TOOL WALK-THROUGH

In this section, we shall demonstrate our developer's debugging tool for defect detection and patch suggestion. Our framework has been embedded into an Eclipse Plugin (*EnergyDebugger*) that facilitates easy access for developers. As shown in Figure 5, the debugging process begins when the developer selects an open project in *EnergyDebugger* perspective. At this point, the selected project is parsed to obtain project metadata such as class description, method signatures, method-to-line number mappings, loop-to-line number mappings, etc. The framework also keeps track of `Activities`, `Services` and `Listeners` within the app. The code for metadata collection is implemented using JDT core (`org.eclipse.jdt.core`) libraries [35]. The debugging process then allows the developer to load a log file which is pre-generated by user, using our instrumentation and logging tool. Following automatic contextual analysis and defect localization, the debugging tool reports all found defects and their associated patch suggestions; each defect can be selected by developers for further visualization and understanding. We present a walk-through of our debugging tool for two Android apps, namely *Ushaihdi Android* [36] and *Shortyz* [27].

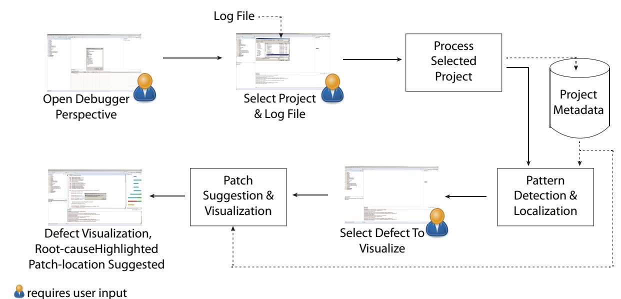



**Figure 5: Working with the debugging tool in Eclipse**

**Table 3: Open-source, Android apps that were used in the evaluation of our framework**

| | App Name version/code | App Description | LoC/Apk Size (KB) | Event Handler Classes | Android API Calls | Energy-intensive Resources Used* |
|---|---|---|---|---|---|---|
| Patched Apps | DroidAR 1.0/1 [21] | An augmented-reality app for Android. | 18177/398 | 6 | 224 | $n, l, c, s, d$ |
| | Osmdroid 3.0.1/2 [22] | Provides replacement for Android's MapView | 8107/276 | 10 | 544 | $n, l, s, d$ |
| | Osmdroid 1.0.0/1 [17] | Provides replacement for Android's MapView | 5308/225 | 9 | 413 | $n, l, d$ |
| | Recycle-locator 1.0/1 [16] | Area-specific restroom, mailbox finding app. | 717/116 | 3 | 61 | $n, l$ |
| | SP Transport 1.08/9 [13] | Android app that assists in bus-travel | 1437/142 | 3 | 23 | $n, l$ |
| | SP Transport 1.17/18 [23] | Android app that assists in bus-travel | 1766/161 | 3 | 56 | $n, l$ |
| | Ushaidi v2.2/13 [24] | App for Collection, visualization for crisis data | 10621/713 | 22 | 276 | $n, l, p, c, d$ |
| Unpatched Apps | Aripuca 1.3.4/24 [25] | Recording tracks and saves waypoints. | 8093/660 | 14 | 730 | $n, l, s, d$ |
| | Benchmark 1.1.5/9 [26] | Comprehensive benchmark suite for Android devices | 9739/1020 | 23 | 71 | $n, p, d$ |
| | Shortyz 3.1.0/30100 [27] | App to downloads and displays crossword puzzles | 5638/175 | 12 | 568 | $n, d$ |
| | iTLogger 1.0.0 / 2 [28] | An app for measuring road quality using on-board sensors | 4014/553 | 7 | 205 | $l, s, b, c, p, d$ |
| | Omnidroid 0.2.1/6 [29] | Automated event/action manager for Android | 12425/258 | 28 | 393 | $n, l, c, d$ |
| | MobiPerf 2.5/1050 [30] | App for doing mobile network measurements. | 8009/401 | 12 | 340 | $n, l, p$ |
| | Sensorium 1.1.8/11 [31] | Collects sensor data such as 3G, GPS, battery charge. | 4001/1248 | 6 | 7840 | $n, l, b, s, d$ |
| | StrobeLight 1.2/3 [32] | Strobe light apps using camera-flash | 210/22 | 1 | 21 | $c$ |
| | Userhash 1.1/2 [33] | View location of friends and family. | 837/171 | 7 | 405 | $n, l, p$ |
| | Zmanim 3.3.84.296/84 [34] | List of halachic/halakhic times. | 72977/842 | 4 | 1102 | $n, l$ |

* Network($n$), Location($l$), Camera($c$), Sensors($s$), Storage($d$), Power Management($p$), Bluetooth($b$)

*Ushaihdi Android* is an open source app for crowd-sourcing crisis information over the internet. In our experiments we use version $v2.2$ of the app, which had a known vacuous background services defect (Issue 11). Figure 4(a) presents *some* of its GUI elements. The test case obtained from the online issue report to generate the faulty behaviour is `App Start` → `OK` → `Back` → `Add Report` → `Home`. Figure 4(c) shows a partial screenshot of our tool for debugging *Ushaihdi Android*. The screen shows the scenario when a defect of vacuous background services is found and visualized. The editor view on the left displays the source code `UserLocationMap.java`, where the `setDeviceLocation` method is highlighted. The image view on the right visualizes the defect scenario, including the *when*-context and the *how*-context, in a sequential order, and two service acquisition events are highlighted in yellow. The user may click any of those nodes in the image to jump to their respective invoked source code in the editor. The pop-up message, together with all the highlights, gives users a clear clue for patching. For this app, the suggested patch involves stopping the service in the `onPause` method of class `UserLocationMap.java`.

Note that the defect scenario is derived from its corresponding contextual profile subgraph, as shown in Figure 4(b), via contextual analysis. Each evidential method node in the visualization contains a detailed method signature, which can easily redirect each method node to its invoked source location. In addition, our debugging tool provides different hierarchical views (e.g., the contextual subgraph) for users to understand defects; for the sake of concise presentation, we show a simple image view in Figure 4(c).

```
Downloaders.java
203  HashSet<File> newlyDownloaded = new HashSet<File>();
204
205  for (Downloader d : downloaders) {
206      d.setContext(context);
207
208      try {
209          String contentText = "Downloading from " + d.getName();
210          Intent notificationIntent = new Intent(context, PlayActivity.class);
211          PendingIntent contentIntent = PendingIntent.getActivity(context, 0,
212                  notificationIntent, 0);
213          not.setLatestEventInfo(context, contentTitle, contentText, contentIntent);
214
215          if (!this.supressMessages && this.notificationManager != null) {
216              this.notificationManager.notify(0, not);
217          }
218
219          File downloaded = new File(crosswords, d.createFileName(date));
220          File archived = new File(archive, d.createFileName(date));
```

**Figure 6: Defect localization for the *Shortyz* app**

Our framework also performs an additional step of processing which could not be highlighted in the previous example. This step checks if the detected defect patterns falls within a loop (*i.e.* the scenario of loop-energy hotspot) or within a region of code that is launched on reboot (*i.e.* the scenario of immortality bug). We describe this step by using the *Shortyz* app. *Shortyz* is a crossword puzzle application that downloads, processes and displays free crossword puzzles from a variety of internet locations. After obtaining the log-files our framework generates the profile call graph (see Figure 2(b)). To achieve its functionality, this app launches puzzle download procedure repeatedly (method `download` in class `Downloaders.java`). These procedures fetch puzzle data over the network (see invocations of APIs, *URL* and *HttpGet*, in Figure 2). The interesting insight which our framework provides is that a processing step (method `processDownloadedPuzzle` in class `Downloaders.java`) is triggered in between two download procedures, leading to occurrence of tail-energy hotspot in the app. Additionally, the download-processing iterations happen multiple times in a loop, as a result this app has an instance of loop-energy hotspot. Our framework precisely locates the defect location (see Figure 6, where line 205 indicates the loop) and provides all these information (through highlighting, graphs and reports) to assist the developer in fixing the defect.

# 4. EVALUATION

We evaluated our framework to address the following research questions: (*i*) Does the log-based *field failure* localization approach, that has been used in our framework, works for real-life apps? Is this approach scalable? (*ii*) How useful are the results of defect-localization approach and the patch-suggestion approach, that are used in our framework? (*iii*) Is it possible to reduce the energy-consumption of the affected apps by applying the patches suggested by our framework? If so, by how much?

## 4.1 Experimental Setup

We used an off-the-shelf Samsung S4 smartphone to generate log-files for our experiments. The specific device that was used in our experiments was equipped with a Quad-core 1.6 GHz CPU, 2 GB RAM and was running Android Kitkat OS (4.4.2). The instrumentation and logging utility, debugging framework, as well as a power measurement utility were all run on a PC with an Intel Core i7 processor, 8 GB RAM and Windows 7 OS. We used the setup shown in Figure 7 to measure the energy consumption of an app on the mobile device. In particular, we used a Monsoon Power Monitor [37] to supply the mobile device with a steady voltage of 4.2 Volts and to measure its power consumption. During interactions with the mobile device (when measuring energy-consumption) we

**Table 4: Summary of defect localization and patch location suggestion for *patched-apps***

| App Name version/code | Issue No, Defect Description, Assigned Priority | Issue Open Days | Commits While Issue Open | Observed Changes Class (Method) | Proposed Changes Class (Method) | Energy Saved (%) |
|---|---|---|---|---|---|---|
| DroidAR 1.0 / 1 | Issue 27, Vacuous background services Priority-Medium | 3 | 2 | Commit : $2e315c080974a56751e$ EventManager.java(resumeEventListeners) **Setup.java**(onDestroy,onStop,onRestart killCompleteApplicationProcess,**onPause**) ArActivity.java(onResume), GeoUtils.java(disableGPS) | Setup.java (onPause) | 29 |
| Osmdroid 3.0.1 / 2 | Issue 53, Vacuous background services Priority-Medium | 240 | 559 | Commit : $r751$ **SampleMapActivity.java**(onResume,**onPause**) | SampleMapActivity.java (onPause) | 11 |
| Osmdroid 1.0.0 / 1 | Issue 76, Expensive background service Priority-Medium | 11 | 4 | Commit : $fd2b17227ab183a5a16$ **MyLocationOverlay.java** (getLocationUpdateMinTime, getLocationUpdateMinDistance,**enableMyLocation**, setLocationUpdateMinDistance) | MyLocationOverlay.java (enableMyLocation) | 5 |
| Recycle-locator 1.0 / 1 | Issue 33, Vacuous background services Priority-Medium | 0 | 0 | Commit : $f19b7fc5a7a0$ **Map.java**(**onPause**, createMap,locateUser) | Map.java (onPause) | 23 |
| SP Transport 1.08 / 9 | Issue 38, Sub-optimal resource binding Priority-Medium | 2 | 9 | Commit : $698a27e83900e42a6dd$ LocationView.java(disableLocationUpdates,onResume, enableLocationUpdates) **HtmlResult.java** (**showResult**) | HtmlResult.java (showResult) | 17 |
| SP Transport 1.17 / 18 | Issue 76, Vacuous background services Priority-High | 0 | 0 | Commit : $d2dfa786da728ae6975$ **LocationView.java**(**onPause**, onCreateDialog) | LocationView.java (onPause) | 12 |
| Ushaidi v2.2 / 13 | Issue 11, Vacuous background services Priority-n/a | 0 | 0 | Commit : $561d6fb10a5fc600ab6$ **UserLocationMap.java** (**onPause**, onDestroy) | UserLocationMap.java (onPause) | 10 |

followed a few timing protocols to maintain consistency across all measurements. For instance, the interval between two successive events (touches/taps/clicks) in the test case was 15 seconds and an idle time of 45 seconds was observed just after the app had stopped execution. Also for all experiments the screen time-out duration of the device was set to 15 seconds.

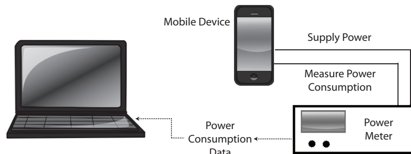



**Figure 7: Power-measurement setup**

## 4.2 Subject Programs

We gathered two sets of open-source apps to be used in our experiments. We shall henceforth refer to these sets as *patched-apps* and *unpatched-apps*. *Patched-apps* consist of apps for which there exist known energy-related issues reported by the user (along with test-scenarios) and patches that are applied by the developers to resolve the energy-related issue. *Unpatched-apps* on the other hand consist of apps for which energy-related issues were known (from previous works such as [6]) but there were no developer-provided patches to fix the issue. Table 3 provides some key information for both sets of apps. These apps are diverse in size and use of energy-intensive resources. The number of event-handler classes (*i.e.* related to activities, services and receivers, details in Table 1) varied from 1 to 28, whereas the number of Android API calls related to energy-intensive resources varied from 21 to 7,840. The energy-intensive resources used by each app are shown in the last column of Table 3. The line-of-code (LoC) of these apps varied between 201 to 72,977 (average LoC of 10,122). The links for source-code of subject apps are provided in [38].

## 4.3 Efficacy of Defect-detection

We conducted the first set of experiments with the *patched-apps*. These apps were specifically useful in the initial stages of this work because they provided us with useful insights into how users observed and reported energy-related issues, as well as, how developers analysed and fixed these issues. A list of these issues can be found in column 2 of Table 4. We observed that when these issues were brought to developer's notice they were classified with *medium* or *high* priorities. Also the developers usually fixed these defects within a fortnight. However, in some cases issues remained open for a considerably long period of time, mostly due to improper understanding of the defect. Consider the Issue 53 [22] in app Osmdroid, which stayed open for 240 days during which time 559 versions were committed to the repository. We present the timeline of the conversation related to Issue 53 in following.

| | |
|---|---|
| May 25, 2010 | Issue opened. Test case provided. |
| Oct 4, 2010 | Priority increased from *low* to *medium* |
| Oct 8, 2010 | User thinks defect is fixed |
| Jan 11, 2011 | Project member suggests closing the issue |
| Jan 12, 2011 | Project member claims the issue is not fixed |
| Jan 20, 2011 | Project member fixes Issue. Provides commit number were issue has been fixed and closes issues. |

Unlike the confusion which ensues as a result of ad-hoc communication, our framework provides definitive answers as to whether there exists a defect for a given test case/scenario. When we tested the log-files generated from the apps (including for Osmdroid Issue 53) our framework was able to pinpoint exact locations of defects in the source-code (manually cross-checked by going through user-comments, code-changes, etc). Also the defect visualization provided by our framework further assists the developer in understanding the cause and effect of the observed defect. Once confident about the efficacy of our framework (in detecting defects), we conducted another set of experiments with the *unpatched-apps* for which our framework was able to pinpoint locations of the defects in the source-code as well. Table 6 provides a list of defect-locations for this set of apps.

## 4.4 Scalability of Defect-detection

The scalability of our technique is mostly dictated by the sizes of the log-files that are used for profile generation. In general, programs with more event-handlers and Android API usages may generate bigger log-files. The length of test-input sequence may also

affect the size of log-files. It is worthwhile to know that LoC may not be a good indicator of log-file size, as it may not correspond to more event-handlers or Android API usages. For example, app *Shortyz* has only $5,538$ LoC but generated $50,049$ lines of log messages whereas app *Benchmark* has $9,739$ LoC but generated only $356$ lines of log messages (*cf.* Tables 5 and 6). In general, the analysis was relatively fast, with the longest analysis time being $3.1$ seconds for $112,897$ lines of log messages for app *DroidAR*.

**Table 5: Line of log messages and analysis time for all apps**

| App Name (version/code) | Line of Log Messages | Analysis Time (seconds) |
|---|---|---|
| DroidAR (1.0 / 1) | 112,897 | 3.1 |
| Osmdroid (3.0.1 / 2) | 3,504 | 0.1 |
| Osmdroid (1.0.0 / 1) | 1,526 | 0.1 |
| Recycle-locator (1.0 / 1) | 396 | 0.1 |
| SP Transport (1.08 / 9) | 543 | 0.1 |
| SP Transport (1.17 / 18) | 14,281 | 0.2 |
| Ushaidi(v2.2 / 13) | 876 | 0.1 |
| Aripuca (1.3.4 / 24) | 3,838 | 0.5 |
| Benchmark (1.1.5/ 9) | 356 | 0.1 |
| Shortyz (3.1.0 / 30100) | 50,049 | 0.6 |
| iTLogger (1.0.0 / 2) | 2,967 | 0.1 |
| Omnidroid (0.2.1 / 6) | 2,284 | 0.2 |
| MobiPerf (2.5 / 1050) | 3,210 | 0.2 |
| Sensorium (1.1.8/ 11) | 11,347 | 0.5 |
| StrobeLight (1.2 / 3) | 87 | 0.1 |
| Userhash (1.1 / 2) | 226 | 0.1 |
| Zmanim (3.3.84.296 / 84) | 50,892 | 0.8 |

## 4.5 Effectiveness of the Patch-suggestion

The final set of experiments (with a power-measurement setup as described in Section 4.1) were conducted to evaluate the effectiveness of the patch-suggestion. For the *patched-apps* this was relatively straightforward as we could compare the patches suggested by our framework to the patches applied by the developer for each defect. However, for *unpatched-apps*, since there was no such information available, we compared energy consumption before and after patches were applied, to evaluate the effectiveness of the patches. For the experiments with *patched-apps*, we observed a significant correlation between the changes suggested by our framework and changes added to the source-code commits where the defect was resolved (*cf* Table 4, columns 5 and 6). Power measurement experiments with the *unpatched-apps* also returned positive results, with energy-savings varying from 6% to 29%.

**Table 6: Summary of results for *unpatched-apps***

| App Name version/code | Defect Description | Defect Location | Energy Saved (%) |
|---|---|---|---|
| Aripuca 1.3.4/24 | Vacuous services | AppService.java( startLocationUpdates, startSensorUpdates) | 15 |
| Benchmark 1.1.5/9 | Wakelock Bug | Benchmark (onCreate) | 29 |
| Shortyz 3.1.0/30100 | Loop-energy hotspot | Downloaders.java( processDownloadedPuzzle, download) + 3 more | 11 |
| iTLogger 1.0.0/2 | Vacuous services | iTloggerActivity (onCreate) | 9 |
| Omnidroid 0.2.1/6 | Immortality Bug | LocationMonitor (init) | 14 |
| MobiPerf 2.5/1050 | Wakelock Bug | PoneUtils (acquireWakelock) | 12 |
| Sensorium 1.1.8/11 | Vacuous services | GPSLocationSensor(_enable), NetworkLocationSensor(_enable) | 21 |
| StrobeLight 1.2/3 | Sub-optimal binding | StrobeRunner (run) | 6 |
| Userhash 1.1/2 | Immortality Bug | ViewActivity (onClick) | 15 |
| Zmanim 3.3.84.296/84 | Resource Leak | LogUtils (startSession) | 21 |

# 5. RELATED WORK

[*Monitoring*] Due to increasing popularity of smartphones and their limited battery power, energy consumption models of mobile apps have been extensively studied [7, 39–45] for monitoring and optimizing their energy usage. These models, typically based on monitoring device components' measurable parameters and computing the approximate battery drain caused by the component over time, provide an energy-aware quality of service support for users.

[*Profiling*] Increasing attention has been drawn to the research connecting energy consumption models with various user or system activities in mobile applications. For examples, real user activities [46, 47] were collected and studied to guide power consumptions for mobile architectures. An accurate power model was constructed based on both the power management and activity states of those power-intensive hardware components [39]. Eprof [44,48] is a fine-grained energy profiler for smartphone apps, mapping the power draw and energy consumption to program entities via a Android API driven finite state machine [44]. PowerScope [49] is an alternative tool for profiling the energy Usage of Mobile Applications. However, those energy profiling tools still focus on answering the ultimate question, "*Where is the energy spent inside my app?*", to provide an energy-aware quality of service support.

[*Diagnosing*] Several automated tools [50–52] have been recently developed for detecting energy anomalies and diagnosing energy inefficiency for mobile applications. The ADEL tool [50] detects and isolates energy leaks resulting from unnecessary network communication via data flow analysis. The Carat tool [51] detects and diagnose energy anomalies by using a collaborative battery consumption and utilization measurements aggregated from multiple clients. The GreenDroid tool [52] monitors sensor and wake lock operations to detect two common causes of energy problems: missing deactivation of sensors or wake locks, and cost-ineffective us of sensory data, and generate detailed reports to assist developers in validating detected energy problems.

[*Debugging*] Debugging functionality-related defects, e.g., fault localization [2,53] and patch generation [1,3,4], is difficult and time-consuming. Debugging energy-inefficiency defects is even more challenging because the defects may depend on the running contextual sensitive scenario, including the state of hardware device, a specific sequence of user interactions, and program call dependencies. For locating energy-inefficiency issues, developers often seek assistances from users through online coding repositories since no debugging tool is currently available. Our framework in this paper provides both an effective communication channel from users to developers and an automated energy-inefficiency debugging tool.

# 6. CONCLUSION

We present a practical framework for localizing energy-efficiency related *field failures* in mobile apps. Our framework provides a simple yet effective communication protocol to be used between the users and developers. Not only does our developer-side debugging tool detect and localize the presence of energy defects in the app source code, but also suggests potential patch location for the reported defects. Evaluation with real-life apps have shown that our tool can localize defects effectively and efficiently, even for apps with thousands of lines-of-code. Additionally, the energy savings generated as a result of the patched defects are significant (observed energy saving between $5\%$ to $29\%$).


# ACKNOWLEDGEMENT

The work was partially supported by a Singapore MoE Tier 2 grant MOE2013-T2-1-115 "Energy aware programming"; Dr. Hai-Feng Guo is currently visiting NUS, sponsored by the grant.

## 7. REFERENCES


[1] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, 2009.

[2] W. Eric Wong and Vidroha Debroy. Software fault localization. In *Encyclopedia of Software Engineering*, pages 1147–1156. 2010.

[3] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.

[4] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[5] Mytracks, issue 520. `https://code.google.com/p/mytracks/issues/detail?id=520`.

[6] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

[7] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, 2009.

[8] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, 1982.

[9] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2), April 1998.

[10] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 2004.

[11] Mikhail Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, 2004.

[12] Csipsimple, issue 81. `https://code.google.com/p/csipsimple/issues/detail?id=81`.

[13] Sofia public transport, issue 38. `https://github.com/ptanov/sofia-public-transport-navigator/issues/38`.

[14] Adw-launcher-android, issue 202. `https://code.google.com/p/adw-launcher-android/issues/detail?id=202`.

[15] Google-voice-locaton, issue 4. `https://code.google.com/p/android-google-voice-locations/issues/detail?id=4`.

[16] Recycle-locator, issue 33. `https://code.google.com/p/recycle-locator/issues/detail?id=33`.

[17] Osmdroid isuue, 76. `https://code.google.com/p/osmdroid/issues/detail?id=76`.

[18] The Android Open Source Project. Adjusting the model to save battery and data exchange. http://developer.android.com/guide/topics/location/strategies.html.

[19] Omnidroid issue, 98. `https://code.google.com/p/omnidroid/issues/detail?id=98`.

[20] K9mail issue, 424. `https://code.google.com/p/k9mail/issues/detail?id=424`.

[21] Droidar, issue 27. `https://code.google.com/p/droidar/issues/detail?id=27`.

[22] Osmdroid, issue 53. `https://code.google.com/p/osmdroid/issues/detail?id=53`.

[23] Sofia public trasport, issue 76. `https://github.com/ptanov/sofia-public-transport-navigator/issues/76`.

[24] Ushahidi, issue 11. `https://github.com/ushahidi/Ushahidi_Android/pull/11`.

[25] Aripuca gps tracker. `https://f-droid.org/wiki/index.php?title=com.aripuca.tracker`.

[26] 0xbenchmark. `https://f-droid.org/wiki/page/org.zeroxlab.zeroxbenchmark`.

[27] Shortyz. `https://f-droid.org/wiki/page/com.totsp.crossword.shortyz`.

[28] itlogger. `https://f-droid.org/wiki/page/de.tui.itlogger`.

[29] Omnidroid. `https://f-droid.org/wiki/page/edu.nyu.cs.omnidroid.app`.

[30] Mobiperf. `https://f-droid.org/wiki/page/com.mobiperf`.

[31] Sensorium. `https://f-droid.org/wiki/page/at.univie.sensorium`.

[32] Strobe light. `https://f-droid.org/wiki/page/com.stwalkerster.android.apps.strobelight`.

[33] Userhash. `https://f-droid.org/wiki/page/com.threedlite.userhash.location`.

[34] Halachic prayer times. `https://f-droid.org/wiki/page/net.sf.times`.

[35] Jdtcore. http://www.eclipse.org/jdt/core/.

[36] Ushahidi. https://www.ushahidi.com/.

[37] Monsoon power monitor. `https://www.msoon.com/LabEquipment/PowerMonitor/`.

[38] Link to subject apps. `http://www.comp.nus.edu.sg/~rpembed/energydebugger/subjectapps.xlsx`.

[39] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, 2010.

[40] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, 2010.

[41] Marius Marcu and Dacian Tudor. Energy consumption model for mobile wireless communication. In *Proceedings of the 9th ACM International Symposium on Mobility Management and Wireless Access*, MobiWac '11, 2011.

[42] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile

systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, 2011.

[43] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 317–328, 2012.

[44] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, 2012.

[45] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[46] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[47] Denzil Ferreira, AnindK. Dey, and Vassilis Kostakos. Understanding human-smartphone concerns: A study of battery life. In *Pervasive Computing*, volume 6696, pages 19–33. Springer Berlin Heidelberg, 2011.

[48] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, 2011.

[49] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, 1999.

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

[51] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14.

[52] Yepang Liu, Chang Xu, S.C. Cheung, and Jian Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *Software Engineering, IEEE Transactions on*, 40(9):911–940, 2014.

[53] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, 2011.