Automated Re-factoring of Android Apps to Enhance Energy-efficiency

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

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

ABSTRACT

Mobile devices, such as smartphones and tablets, are energy constrained by nature. Therefore, apps targeted for such platforms must be energy-efficient. However, due to the use of energy oblivious design practices often this is not the case. In this paper, we present a light-weight re-factoring technique that can assist in energy-aware app development. Our technique relies on a set of energy-efficiency guidelines that encodes the optimal usage of energy-intensive (hardware) resources in an app. Given a prototype for an app, our technique begins by generating a *design-expression* for it. A *design-expression* can be described as a regular-expression representing the ordering of energy-intensive resource usages and invocation of key functionalities (event-handlers) within the app. It also generates a set of *defect-expressions*, that are *design-expressions* representing the negation of energy-efficiency guidelines. A non-empty intersection between an app's *design-expression* and a *defect-expression* indicates violation of a guideline (and therefore, potential for re-factoring). To evaluate the efficacy of our re-factoring technique we analysed a suite of open-source Android apps using our technique. The resultant re-factoring when applied, reduced the energy-consumption of these apps between 3 % to 29 %. We also present a case study for one of our subject apps, that captures its design evolution over a period of two-years and more than 200 commits. Our framework found re-factoring opportunities in a number of these commits, that could have been implemented earlier on in the development stages had the developer used an energy-aware re-factoring technique such as the one presented in this work.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques

Keywords: Mobile Apps; Re-factoring; Energy-efficiency

1. INTRODUCTION

Easy access to app-development tools and a low barrier to entry[1] has led to an abundance of mobile apps in recent days. As of year 2015, there were more than 1.8 million apps available on Google Play Store [2] alone. A plethora of online tutorials and publicly available testing tools, such as MonekyRunner [3], make it relatively easy even for new app-developers to develop and test the functionality of their apps. However, the same cannot be said for testing non-functional behaviour, specifically energy-efficiency. Mobile devices are energy-constrained by nature. Therefore, it is crucial that apps that are made for such devices be designed and optimized for energy-efficiency. However, due to a combination of factors such as lack of proper understanding of energy-efficient designs or lack of tools that can enforce such energy-efficient designs, app development has mostly been done in an energy-oblivious manner.

In recent years, research works have proposed a number of techniques (such as profiling [4], testing [5]) that can be used post development for quality assurance purposes. Such techniques however do not provide adequate support for energy-efficient design and re-factoring of mobile apps. In this paper, we present an orthogonal approach to address this issue. We present a light-weight, re-factoring technique that uses a set of energy-efficiency guidelines to generate energy-efficient re-factorings for a given app. These energy-efficiency guideline were formulated under the assumption that energy-efficiency can be increased by optimizing the usage of energy-intensive (hardware) resources. Resources such as I/O Components and power management utilities have the biggest impact on energy-consumption, hence their usage must be reduced as much as possible without affecting the functionality of the app. Additionally, certain resources (such as sensors) can be accessed through multiple configurations, each of which provide specific trade-offs between Quality-of-Service and energy-efficiency. Judicious usage of less-expensive resources, based on the functionality of the app, can further decrease energy consumption.

To detect re-factoring opportunities, our framework checks for violations of (energy-efficiency) guidelines in a given app. However, doing so directly on the app source-code may be in-appropriate for a number of reasons. For instance, mobile apps being event-driven in nature, usually consists of segregated pieces of code (or event-handlers), ordering between which may not be explicit in the app-source code. This makes it difficult to detect guideline violations across event-handlers boundaries. Additionally, real-life apps may contain thousands of lines of code, not all of which affect the energy-consumption behaviour of the app significantly. Therefore, before our framework looks for re-factoring opportunities, it first generates an intermediate, *succinct* representation of the app. This intermediate representation, henceforth referred to as *design-expression*, contains only that information which is most relevant to the energy consumption behaviour of the app. More formally, a *design-expression* can be described as a regular expression that represents the ordering of energy-intensive resource usages and invocation of key functionalities (event-handlers) within the app. The use of *design-expression* allows us to re-factor energy-intensive resources across event-handler boundaries. Additionally, since *design-expression* are customized regular expression we can

[1]Registration for a publisher account at Play store costs $25 [1]

MobileSoft'16, May 16-17, 2016, Austin, TX, USA

http://dx.doi.org/10.1145/2897073.2897086 .

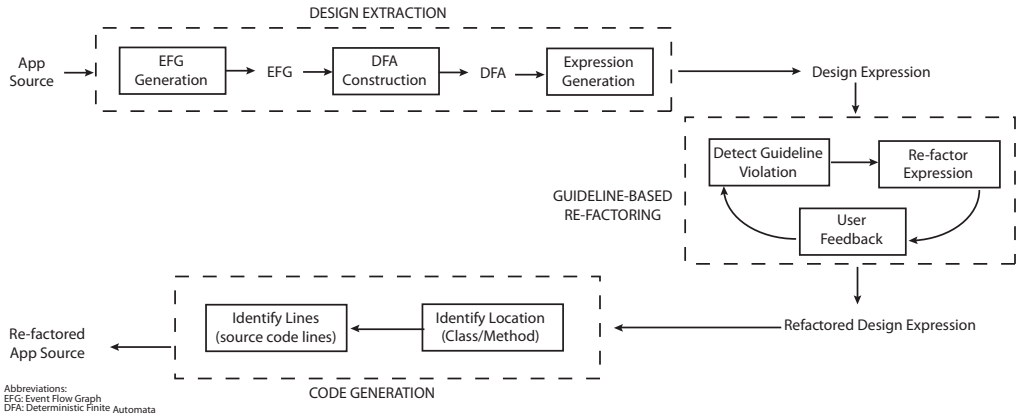


Figure 1: An overview of our framework

use off-the-shelf tools and techniques to analyse/manipulate them. It is also worthwhile to know that our framework generates the *design-expression* for a given app *automatically*.

In order to detect guideline violations our framework also generates a set of *defect-expressions*. A *defect-expression* has same syntax as that of *design-expression* but represent the negation of an energy-efficiency guideline. So essentially, *design-expression* represents what an app is supposed to do (in order to achieve its functionality) whereas the *defect-expression* represents what an app is *not* supposed to do in order to be energy-efficient. A non-empty intersection between *design-expression* and *defect-expression* indicates violation of the energy-efficiency guideline that is associated with the *defect-expression*. It is worthwhile to know that such an analysis is possible because both the *design-expression*, as well as the *defect-expression* are constructed from the same alphabet. On detecting a guideline violation, our framework generates a re-factored *design-expression* such that it has an empty intersection with the *defect-expression*. Finally, the re-factored *design-expression* is presented to the app-developer for approval. If the developer approves the presented re-factoring, the changes are mapped back to the source code.

Key Contributions:

- We present a set of energy-efficiency guidelines that are specifically designed for Android apps. These guidelines can improve the understanding of energy-efficient design patterns amongst app-developers and also provides the groundwork necessary for construction of energy-aware re-factoring tools.
- We present an automated framework, that can detect and re-factor energy-efficiency guideline violations in Android apps. The efficacy and scalability of our framework comes from the use of a novel intermediate representation of a mobile-app *i.e. design-expression*. These *design-expression* contains only that information which is crucial for improving the energy-consumption behaviour of the app, while maintaining its original functionality.
- We evaluated our framework with a suite of open-source applications from the F-Droid [6] app repository. In this evaluation, we observed that the re-factorings generated by our framework improved the energy-consumption of the evaluated apps significantly (observed improvements 3% to 29%).
- We also present an in-depth case study for one of our subject apps, that captures its design evolution over a period of two-years and more than 200 commits. Our framework found re-factoring opportunities in a number of commits, that could have been implemented earlier on in the development stages, had the developer used an energy-aware re-factoring technique such as the one presented in this work.

2. OVERVIEW

Our framework is composed of three key components (overview shown in Figure 1): (i) design extraction component (ii) re-factoring component and (iii) code generation component. The objective of design extraction component is to generate the design-expression for the app. The most crucial processing happens in the re-factoring component, where the design-expression is evaluated for guideline violation and design expression re-factoring takes place (if any guideline violations are detected). Finally, the code generation component maps the changes from the re-factored design-expression to the app source code. These components are discussed in detail in sections 2.2 - 2.4, with the help of an example-app that is described in Section 2.1.

2.1 Example App

To keep the proceeding discussion concrete, we shall explain the overview of our framework using an example-app. Let us consider a simple app that allows its user to search for famous landmarks based on provided keywords. If the user selects any of the landmarks, the app shows the landmark on a map, along with the distance of the device to the selected landmark. The app has local copies of all the information (landmark names, co-ordinates, map tiles, etc) that is required to do its computation, except for the user/device location. The user/device location is obtained through an on-board GPS receiver. The app initiates the location updates as soon as it is started and the location updates are stopped only when the app exits (the foreground). The screen-shots provided in Figure 3 can provide a rough idea about the graphical user interface (GUI) layout of the app. It is worthwhile to know that location-updates are one of the most energy-intensive operation on a mobile device and hence it should be used for as small duration of time as possible. However, in this example-app location-updates have been used sub-optimally. More specifically, the location updates are active for the entire duration of time the app is active (in the foreground), whereas the location-updates are used only when the user selects a landmark (*i.e* when Screen 2 is shown). Through our framework we wish to detect and re-factor instances of energy-inefficient behaviour, such as sub-optimal resource binding as present in this example-app.

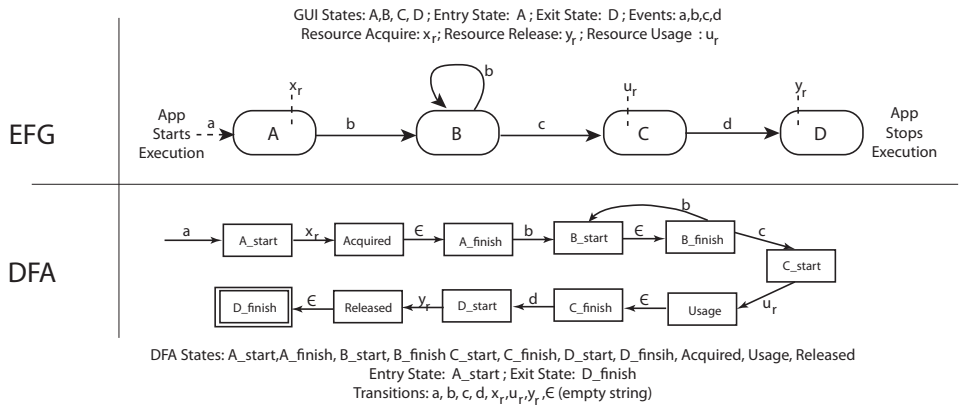
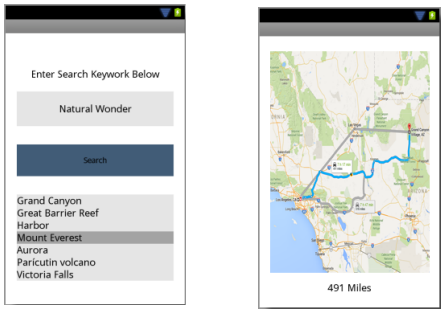


Figure 2: Event-flow graph (EFG) and deterministic finite automata (DFA) for the example-app of section 2.1

2.2 Design Extraction



Screen 1 Screen 2

Figure 3: An example app

Our framework begins analysis by generating an appropriate intermediate representation of the app that is to be evaluated. Performing re-factorings directly on the app source code would be inappropriate for a number of reasons. For instance, mobile apps being event-driven in nature, usually consists of segregated pieces of code (or event-handlers), ordering between which may not be explicit in the app-source code. This makes it difficult to detect energy-inefficient patterns across event-handlers. Additionally, real-life apps may contain thousands of lines of code, not all of which affect the energy-consumption behaviour of the app significantly. Therefore, we create a succinct intermediate representation of an app which contains only the information that is most relevant to its energy consumption behaviour.

Energy consumption in mobile apps has a direct co-relation to the use of Android API calls that are related to acquire, usage and release of energy-intensive resources [4]. Previous works such as [7] have found that the Screen, Wifi, GPS, Sensors, Camera, CPU and Keypad are some of the most energy-intensive resources on a mobile device. Hence, acquire, usage and release API calls for these energy-intensive resources are included in the intermediate representation. Additionally, the intermediate representation also captures the different user-interaction patterns by which an user can interact with the app. Our objective, after all, is to re-factor an app so as to remove (or at least minimize) the user interaction (UI) patterns that may lead to energy-inefficient behaviour. Considering all these requirements we create the notion of *design-expression*.

DEFINITION 2.1. *A design-expression is a regular expression which represents the ordering of Android API calls (acquires, release & usages) for energy-intensive hardware resources and invocation of event-handlers within an app.*

A design-expression is similar to a regular expression in terms of syntax and expressibility. Like a regular expression, a design-expression is constructed with symbols and operators. The symbols of the expression are user-inputs (such touches, taps, etc) while operators are the same as regular grammar (*eg.* $*$ implies 0 or more). A detailed discussion on regular expression grammar can be found in [8]. The key advantages of using design-expression can be summed up as follows:

- It is a succinct representation for an app and contains only that information which is most relevant to its energy consumption behaviour. It is worthwhile to know that design-expression can be used to represent the set of all input strings that can be used to interact with the app.
- Since design-expressions are based on regular expressions we can use a wide-variety of existing tools and techniques that are applicable to regular expressions, to manipulate design expression (such as minimizing an expression or computing the intersection of two expressions, etc).

Generating of the design-expression from app sources takes place in three steps: (i) EFG Generation, (ii) DFA Construction and (iii) Expression Generation.

***(i) EFG Generation*:** An event-flow graph (EFG) [9] can be used to represent the GUI model of an app and can be defined as in Definition 2.2. Figure 2 shows a simplified EFG for the example-app of section 2.1. The GUI states A and D correspond to app start and exit states. While the states B and C correspond to the app being in Screen 1 and Screen 2, respectively (*cf.* Figure 3). The events a and d represent the starting and closing of the app. Whereas the event b represents the user pressing search button and the event c represents the user selecting a landmark. Since an user can repeatedly press the search button on the screen 1 therefore there is a self-loop at EFG node B. It is worthwhile to know that EFGs for real-life apps can be more complicated because of the omnipresent UIs such as the Back button and the Menu button. However, for the purpose of simplicity we shall not include these UIs (Back and Menu button) in the EFG of Figure 2. Finally, the Android API calls x_r, u_r and y_r represent the acquire, usage and release of resource r, (in the example of section 2.1 it is location updates).

DEFINITION 2.2. *An event-flow graph is a directed graph, that captures all possible event-sequences that can be used to interact*

with an app. The nodes of an EFG represent GUI states. A directed edge between two nodes of an EFG X and Y represents that state Y follows state X. Additionally, nodes of the EFG are annotated with event-handler information associated with their respective events.

In order to generate the EFG we use an automated, GUI exploration tool Dynodroid [10]. Dynodroid uses a publicly-available, Android tool Hierarchy Viewer [11], to obtain the UI layout of an app. It then uses this layout information to progressively explore all the UI states of an app. By extending Dynodroid we can obtain the events as well the directed edges between the GUI states. We also need to obtain the event to event-handler mapping information for EFG generation. This information can either be obtained by modifying the Android platform or instrumenting the apk files. We choose the later because it is more straightforward and maintainable as it need not be re-implemented every time the Android platform is updated. In particular, the instrumentation is done for event-handlers that are defined in the `android.app.activity`, `android.app.service` and `android.content.BroadcastReceiver` packages of the Android framework. We also obtain an event-handler to Android API call mapping for all the energy-intensive resources by statically analysing the bytecode of an app. For instance, invocation of API call `com.google.android.maps.MyLocationOverlay.enableMyLocation` would be recorded as an acquire for the resource GPS in the event-handler where the API call was used. It is also worthwhile to mention that this event-handler to Android API call mapping is done in an object-insensitive manner. For instance, in this location example all invocation of the API call `com.google.android.maps.MyLocationOverlay.enableMyLocation` would be allocated to the same GPS resource. Finally, the event-handler to Android API call mapping is combined with the event to event-handler mapping as shown in Figure 4, to generate the EFG.

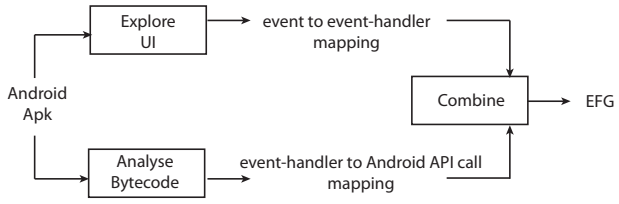


Figure 4: Event-flow graph (EFG) generation

***(ii) DFA Construction*:** Once the EFG is obtained, it is converted into a deterministic finite automata (DFA). This conversion is done so that we can use standard algorithms to do DFA-to-Expression generation. The DFA is constructed such that each node in the DFA either represents the starting of (execution of) an event-handler, stopping of (execution of) an event-handler, acquiring of a resource, release of a resource or usage of a resource. In the scenario where an EFG node is not associated with a resource-related Android API call, the conversion from EFG node to DFA node is straightforward. However, in the scenario where an EFG node does contain resource-related Android API calls, the EFG node is divided into multiple DFA nodes (depending on the number of Android API calls contained in the EFG node). Finally, the entry states and the exit states are copied from the EFG to the DFA. Figure 2 shows the DFA for the example-app of section 2.1.

***(iii) Expression Generation*:** Post DFA construction, we extract the design expression (*i.e.* the regular expression) representing the DFA. The conversion from DFA to expression is done using the standard algorithm as proposed in [8]. Essentially, the algorithm proceeds by removing the DFA states (and changing the transitions accordingly), until only initial and final states are remaining. The resultant expression is subsequently minimized using an off-the-shelf Python library [12]. For instance, the resultant design-expression for the example-app of section 2.1 is $ax_rb^*cu_rdy_r$.

2.3 Guideline-based Re-factoring

The re-factoring component of our framework operates in two steps: detecting guideline violating patterns and re-factoring. To detect guideline-violating patterns in an app's design expression, our framework first generates the defect-expression (for each guideline). A defect-expression can be described as a design-expression representing the negation of a guideline. A non-empty intersection between the (app's) design-expression and defect-expression indicates the presence of a guideline violating pattern. It is worthwhile to know such an analysis is possible because the design expression and defect expression are constructed using the same alphabet.

Consider the example-app from section 2.1 where early-resource binding (for location updates) takes place. Assume that the guideline ϕ represents the fact that resource binding should happen as late as possible, then $\neg\phi$ (defect expression) represents its negation *i.e.* the scenario where early resource binding takes place. The information that there is potentially long delay (on node B) between the acquire and usage of resource r, can be easily obtained through our framework. In particular, for the example-app of section 2.1 design expression and defect expression ($\neg\phi$) are shown in expressions 1 and 2, respectively. Here $\bullet$ implies all feasible symbols and $\neg$ implies negation (of an symbol). Operators $*$ and $+$ represents 0 or more times and 1 or more times, respectively. A non-empty intersection between expression 1 and expression 2 (shown in expression 3), provides an evidence for guideline violation.

$$\text{Design Expression} : ax_rb^*cu_rdy_r \quad (1)$$

$$\text{Defect Expression} : \bullet^*x_r[\neg u_r][\neg u_r]^+u_r[\neg y_r]^+y_r\bullet^* \quad (2)$$

$$\text{Intersection} : ax_rb^+cu_rdy_r \quad (3)$$

$$\text{Re-factored Expression} : ab^*cx_ru_rdy_r \quad (4)$$

In a scenario such as in this example, where guideline-violation is detected, the app's design expression is re-factored such that the resultant design-expression has an empty intersection with the defect expression. The re-factoring method depends on the specific guideline that has been violated (further described in section 3.2), however, following observations can be stated for all re-factored design-expressions.

- *Guideline Conforming*: The intersection between the re-factored design expression and the defect expression is empty. (For example, re-factored design expression shown in expression 4 has an empty intersection with the defect expression shown in expression 2)

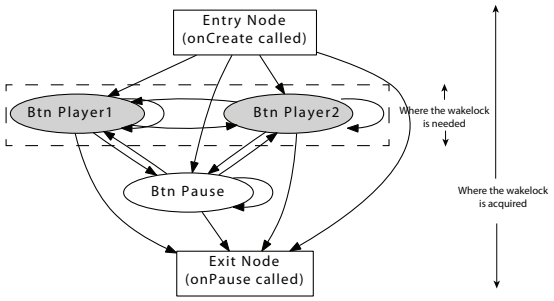
- *Functionality Preserving*: The re-factoring is such that the original functionality is preserved. In particular, re-factoring affects the position of resource acquire and/or release (symbols) in the design expression. However, the resource usage (symbols) are left untouched. Additionally, the relative ordering between resource acquire, usage and release ($acquire \Rightarrow usage \Rightarrow release$) is always ensured.

2.4 Code Generation

Once the design expression has been re-factored and the changes approved by the app-developer, we can map the changes back to the source-code. This is done in two steps: identifying the re-factored

```
1  public void onCreate(Bundle savedInstanceState) {   <- onCreate part of Acitivity lificyle
2          super.onCreate(savedInstanceState);          Called when user starts apps
3          setContentView(R.layout.activity_main);
4          btn1 = (Button) findViewById(R.id.btn1);
5          surfaceView = (SurfaceView) findViewById(R.id.surfaceView);
6          surfaceHolder = surfaceView.getHolder();
7          surfaceHolder.addCallback(this);
8          surfaceHolder.setType(SurfaceHolder.
9                  SURFACE_TYPE_PUSH_BUFFERS);
10         camera = Camera.open();   <- Camera acquired
11         btn1.setOnClickListener(new OnClickListener() {
12                 @Override                  Invoked when user
13                 public void onClick(View v) {  <- clicks Button 1
14                         camera.startPreview();
15                 }                  <- Camera usage started
16         });                        captured video is shown on screen
17 }
18 @Override
19 public void surfaceCreated(SurfaceHolder holder) {
20         try {
22                 camera.setPreviewDisplay(surfaceHolder);
23         } catch (final Exception e) {
24         }
25 }
26 @Override
27 public void surfaceDestroyed(SurfaceHolder holder) {
28         camera.stopPreview();   <- Preview stopped
29         camera.release();
30         camera = null;          <- Camera released
31 }
```

(a)



(b)

Figure 5: (a) A code fragment showing sub-optimal camera binding, (b) Sub-optimal Wakelock acquisition in app ChessClock

location (source-code line numbers within event-handlers) and implementing the changes to new location. As described in the previous section, re-factoring initially happens at the level of design expression, where each symbol in the design-expression corresponds to some event or acquire/usage/release of a resource in the app. It is worthwhile to know that re-factoring only affects the position of resource acquire/release (relative to events) in the design expression. Therefore, by comparing the original design-expression to the re-factored one we can identify the event-handlers that need to be modified. More specifically, two sets of event-handlers are modified: event-handlers where the resource acquire/release Android API call used to be (in the original expression) and event-handlers where the resource acquire/release Android API call need to be (obtained from the re-factored expression). For instance, observe that the resource acquire symbol, x_r, in expression 4 is moved after event c suggesting that Android API call for resource acquire should be moved to the event-handler for invocation of GUI state C (original location of Android API call represented by x_r was in GUI state A). It is worthwhile to know that we record the event-handler to source-code mapping information during the EFG generation step. Therefore, we can easily identify the event-handlers that need to be modified to implement the re-factoring. Additional (syntax-level) information that may be needed to conduct the re-factoring (such as parameters to the Android API call) are obtained through flow-analysis on the original source code.

3. GUIDELINE-BASED RE-FACTORING

The basic premise on which these guidelines for energy-efficiency have been formulated is the fact that minimizing usage of energy-intensive resources increases energy-efficiency of an app. The resources in question being energy-intensive hardware components such as GPS, Camera, Wifi, Bluetooth, Sensors, *etc* or power management utilities such as Android Wakelocks. We shall first discuss the guidelines in section 3.1 and the algorithm for guideline-based re-factoring in section 3.2.

3.1 Energy-efficiency Guidelines

The guidelines can be stated as follow:

1. *Sub-optimal Bindings*: Resources must be acquired as late as possible (during the execution of an app) and released as early as possible.
2. *Nested Usages*: Nesting of resources (acquire-releases) should be avoided.
3. *Trade-offs, QoS Vs Energy-efficiency*: Certain information (such as location updates), can be obtained through multiple resources, each varying in quality-of-service (QoS) and energy consumption. If the application functionality permits, QoS can be traded-off to improve energy-efficiency.
4. *Resource Leaks*: All resources acquired during the execution of app must be released before the app exits.

***Sub-optimal Bindings*:** Roughly translated, this guideline implies that resource acquire, usage and release should be as close to each other as possible. However, due to the event-driven nature of mobile-apps, source-code proximity may not necessarily imply closeness. Consider the code fragment shown in Figure 5(a). This app has the basic functionality to capture images through camera and display them on the screen when user clicks an on-screen button. In this example, the camera is acquired (line 10) when the user starts the app (*i.e.* in function $onCreate$). However, there might be a substantial delay between the resource being acquired and resource being used ($camera.startPreview()$ on line 14). This is because the preview is only started when the user clicks the button, thereby triggering the event handler defined on line 11. The period of time in between the app start and user event is the time when the camera is consuming energy needlessly and can be avoided. It is worthwhile to know that for certain resources, such as Wakelocks, resource usages cannot be explicitly associated with any Android API calls (*i.e.* such resources only have acquire and release API calls). For such cases, developer help may be needed to identify the functionalities (event-handlers) that utilize the resource. For instance, as shown in the Figure 5(b), the app ChessClock [13] requires the Wakelock to be acquired when either of the two players are interacting with the app. However, in the app, Wakelock is acquired for entire duration of app activity.

$$\text{Defect Expression} \quad : \quad \bullet^* x_r[\neg u_r][\neg u_r]^+ u_r[\neg y_r]^+ y_r \bullet^* \quad (5)$$

To generate the defect expression we identify and use the symbols associated with the resource acquire, usage and release in the design expression. Expression 5 shows an example defect expression representing sub-optimal binding for the example-app of Section 2.1, where $acquire(r)$, $usage(r)$ and $release(r)$ for resource r are denoted by symbols x_r, u_r and y_r, respectively. In the scenario, where the intersection between defect expression and design expression is non-empty, the design expression is re-factored. The re-factoring is such that the (symbols for) resource acquire/release

are re-arranged to be as close to resource usage in the design expression. However, during the re-factoring relative ordering, between the acquire, usage and release is always maintained.

***Nested Usages*:** As stated by guideline *Sub-optimal Binding*, resources must be in the acquired state for the smallest period of time possible to achieve the app functionality. Complementary to this guideline is *Nested Usage* guideline, which states that nesting of resources should be avoided. In particular, this guideline applies to those resources which generate same (type of) information. The utility of this guideline is that, if at any stage during re-factoring nesting of resource usages is observed in the design expression, the expression can be simplified by removing the nesting so as to reduce the duration of time for which the acquired resource is active. For example, in the code fragment shown in Figure 6, from app *Sensorium* [14] (commit hash:$9d141b7$), the API call $requestLocationUpdates$ is invoked twice. However, since both the invocations of API call provide location updates, these two invocations can be merged into one, without loss of functionality.

```
1 public void enable() {
2   locationListener = new LocationListener() {
3               //....
4   };
5   locationManager = (LocationManager) context.getSystemService(Context.LOCATION_SERVICE);
6   locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0,  locationListener);
7   locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, locationListener);
8   enabled = true;
9 }
```

Figure 6: A code-fragment from *Sensorium* showing nested resource usage

In expression 6 we show an example of defect expression generation for detecting the presence of nested usage scenarios. Expression 6 is constructed in context of example-app from Section 2.1, where $acquire(r)$ and $release(r)$ for resource r are denoted by symbols x_r and y_r, respectively.

$$\text{Defect Expression} \quad : \quad \bullet^* x_r [\neg y_r]^* x_r \bullet^* \tag{6}$$

***Trade-offs, QoS Vs Energy-efficiency*:** Mobile app functionality is often based on sensor information (such as acceleration, orientation, *etc*) collected from the physical environment. To obtain this information an app must interact with the available I/O components. Since mobile OSs, such as Android, were designed to run on energy-constrained device, they provide a number of ways (API configurations; *cf* Figure 7) to interact with these devices. For instance, in Android most of the power-hungry hardware components (GPS, sensor, screen, etc), can be operated at various levels of power consumption and Quality-of-Service (QoS). In this context, higher QoS implies more precise data, at higher update-frequency. In general, higher QoS leads to higher power consumption.

Table 1: Configuring resources for different QoS, energy-efficiency

Power Cons.	Location Updates (GPS/Network)	Sensor Updates (accelerometer, orientation, etc)	Wakelocks (Screen, CPU, Keypad)
High	gps_provider	sensor_delay_fastest	full_wake_lock, screen_bright_wake_lock
Moderate	network_provider	sensor_delay_game	screen_dim_wake_lock
Low	passive_provider	sensor_delay_normal, sensor_delay_ui	partial_wake_lock

Table 1 shows a list of a few such configurations that can be used in combination with Android API calls to obtain desired QoS. For instance, column 1 of Table 1 lists three different variations of location updates that can be used along with API call $requestLocati$ $onUpdates$. Both $gps\_provider$ and $network\_provider$ actively initiates a location fix, when invoked. $passive\_provider$ on the other hand does not initiate a location fix actively but provides a location fix by passively listening to location updates from other apps (on the device). As a result, it has a significantly less impact on the energy-consumption. However, since the information generated by $passive\ \_provider$ can be stale it is only suitable for apps that require a rough estimation of user's location (for e.g. news apps). Similarly, when comparing between $gps\_provider$ and $network\_provider$, the former provides more precise location update (suitable for travel apps), whereas later is less-precise but relatively more energy-efficient (suitable for continuous location based content generation). It would be impractical to use one set of configuration for all apps, however, once the app-developer provides the app-category (such as used in Play store [15]), appropriate energy-efficient re-factoring can be suggested. In our framework, we look for such configuration in the app source code, with the help of Apache Lucene [16] search libraries.

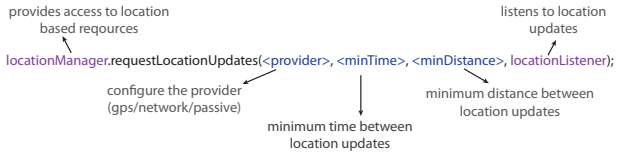


Figure 7: Various parameter that affect QoS, energy-consumption for location updates

***Resource Leaks*:** This is one of most commonly occurring and (energy) expensive defects in mobile apps. Essentially, any resource acquired during the execution of an app must be released before that app ceases to execute. However, since real-life apps have many potential exit locations, ensuring resource releases at all such exit locations may be challenging (specially for apps with many GUI screens). In the presence of such a defect, the device (more specifically the unreleased resource) keeps consuming energy even after the defective app ceases to execute. It is possible to view the scenario of *resource leaks* as a very extreme case of *sub-optimal resource binding*, where the acquired resources are released after infinite period of time. However, we categorize them in two different categories so as to keep the analysis straightforward. For an app to have a resource leak there should be at least one path within the app triggered by a sequence of user-events that ends with an unreleased resource. For instance, defect expression representing resource leaks for the example-app of Section 2.1 is shown in expression 7, where $acquire(r)$ and $release(r)$ for resource r are denoted by symbols x_r and y_r, respectively.

$$\text{Defect Expression} \quad : \quad \bullet^* x_r [\neg y_r]^* \tag{7}$$

3.2 Guideline Implementation

An important question that may arise at this point is how to enforce the energy-efficiency guidelines described in Section 3.1. The two approaches that we can think of for implementing these guidelines would be to either embed them in the platform itself (by means of OS manipulation, middlewares, *etc*) or to enforce them through energy-aware re-factoring tools that assist the app-developer during the development process. It is worthwhile to know that the *middleware approach may be unsuitable for real-life apps as it may make the platform inflexible*. This is because the functionalities of real-life apps may vary widely and are usually subjected to developer discretion. For instance, faster, energy-hungry sensor updates are unsuitable for battery life (such as stated in QoS Vs energy-efficiency trade-offs guideline), however, the app-developer may

still want to use it. Similarly, all acquired resources should be released before the (resource acquiring) app leaves the foreground (as stated in resource leak guideline), but the app-developer may choose not to do so (say for instance the app wants to log the user whereabouts throughout the day using location-updates). In contrast, our *re-factoring framework approach is much more flexible as it allows the developer to choose which of the suggested re-factoring are to be applied* to the source code. Additionally, since all the changes are made only to the app and not to the platform, *the finished app should behave similarly across all devices* (OS/middleware is usually customized by the device vendor).

Algorithm 1 Re-factoring design expression

1: **Input:**
2: App: App source files
3: $AppCt$: Category such as Games, Travel, Books, *etc*
4: **Output:**
5: Ref_{app}: Re-factored design expression
6:
7: $Des_{app} \leftarrow$ `GenerateDesign(`App`)`
8: $\langle v, r \rangle \leftarrow$ `CheckViolation(`$Des_{app}, App, AppCt$`)`
9: **while** $v \in V$ **do**
10: $\langle X_r, U_r, Y_r \rangle \leftarrow$ `GetResSymbols(`Des_{app}, r`)`
11: **if** $(v = SubOptimalBinding)$ **then**
12: $Ref_{app} \leftarrow$ `insertBefore(`Des_{app}, U_r, X_r`)`
13: $Ref_{app} \leftarrow$ `insertAfter(`Ref_{app}, U_r, Y_r`)`
14: **end if**
15: **if** $(v = NestedUsage)$ **then**
16: $Ref_{app} \leftarrow$`merge(`Des_{app}, X_r`)`
17: $Ref_{app} \leftarrow$`merge(`Ref_{app}, Y_r`)`
18: **end if**
19: **if** $(v = TradeOff)$ **then**
20: $Ref_{app} \leftarrow$ `reconfigure(`$Des_{app}, AppCt, X_r$`)`
21: **end if**
22: **if** $(v = ResourceLeak)$ **then**
23: $Ref_{app} \leftarrow$
24: `insertAfter(`Des_{app}, U_r`, {getRelSysCall(`r`)})`
25: **end if**
26: $\langle v, r \rangle \leftarrow$ `getDefect(`$Res_{app}, App, AppCt$`)`
27: **end while**

We use Algorithm 1 to implement the energy-efficiency guidelines that are described in Section 3.1. Algorithm 1 takes in an app and its category as inputs and generates the re-factored design expression, Ref_{app}. It begins by generating the design-expression (procedure `GenerateDesign`) as described in section 2.2. This design-expression is then checked for guideline violations using procedure `CheckViolation`. The procedure `CheckViolation` returns a tuple $\langle v, r \rangle$, where $v \in V \bigcup \{NoDefect\}$ and $V = \{ SubOptimalBinding, TradeOff, NestedUsage, ResourceLeak \}$, whereas, r is the resource that participates in guideline violation v. If a guideline violation is detected (*i.e.* $v \in V$), we proceed to re-factor the design-expression based on the type of guideline violation. To begin re-factoring, we first extract the symbols associated with acquire (X_r), usage(U_r) and release(Y_r) of resource r from the design-expression using the procedure `GetResSymbols`. The procedure for re-factoring depends on the type of guideline violation. In case of suboptimal resource binding guideline violation, the symbols in X_r are inserted (in the re-factored expression) before symbols in U_r. This operation is done using the procedure `insertBefore`. Similarly, procedure `insertAfter` is used to insert symbols in Y_r after symbols in U_r. In case of nested usage guideline violation, the symbols in X_r are merged into a single symbol using the procedure `merge`. Similar merging is done for symbols in Y_r. In case of QoS/efficiency trade-off guideline violation, the design expression stays the same, however, the Android API calls associated with symbols in X_r are re-configured based on provided app category ($AppCt$). Finally, in case of resource leak guideline violation, the symbol for releasing resource r is added after symbols in U_r. The final re-factored expression (Ref_{app}) can be used to map the changes back to the source-code. It is worthwhile to know that Algorithm 1 can re-factor resource acquires/releases across event-handler (class/method) boundaries. This increase the re-factoring opportunities drastically, however, this may also cause some syntax-level inconsistencies (such as due to modifiers associated with variables). However, since our re-factoring technique is made for design/development stages of app-development such inconsistencies can be removed with developer assistance.

```
1 @Override
2 public void onResume() {
3   //...
4   long minTime = 6000;
5   float minDistance = 10;
6   locationManager.requestLocationUpdates(
7           LocationManager.GPS_PROVIDER,
8           minTime, minDistance,
9           locationListener);
10 //...
```

These lines are moved together, if the requestLocationUpdates needs to be re-factored to another location

Figure 8: Re-factoring while maintaining flow-dependencies

It is worthwhile to know that within the source code there may be resource API calls that are dependent on other source code lines for arguments to API calls. To discover and preserve these dependencies we perform flow analysis using the tool Soot [25]. For instance, in the example shown in Figure 8, the invocation of API call $requestLocationUpdates$ (line 6) depends on the value of the long variable $minTime$ (line 4) and float variable $minDistance$ (line 5). In such a scenario, if the re-factoring requires moving the invocation of $requestLocationUpd\ ates$ to another location, all related lines (*i.e.* lines $4 - 9$) are also moved. Currently our framework does not provide support for mitigating challenges that may arise due to aliasing or inter-procedural dependencies. However, our framework can be extended to handle such challenges using the works such as [26]. To identify the re-factoring location (class/method where re-factored lines will be moved/added), we check the position of the resource related symbols with respect to user-event related symbols in the re-factored design expression.

4. EVALUATION

In this section we shall describe the experimental setup and the subject apps (in Section 4.1) and key results of the evaluation (in Section 4.2). Finally, we will present a case study of one of the subject apps, *Sensorium*, in Section 4.3.

4.1 Subject Apps & Experimental Setup

Primarily, we wish to evaluate the efficacy of our technique in detecting the presence of inefficient design-patterns (*cf.* Definition 2.1) and in generating usable energy-efficient re-factoring of the aforementioned patterns in Android apps. To achieve this objective, we create a suite of subject-apps consisting of ten open-source application obtained from the F-droid, open-source Android app repository. These apps [13, 14, 17–24] are diverse in terms of functionality and size (*cf.* Table 2), thereby allowing us to evaluate the different aspects of our framework.

To measure the energy consumption of the mobile device, we created an experimental setup as shown in Figure 9(a). We used the Monsoon Power Monitor [27] to supply the mobile device with a steady voltage of 4.2 Volts and to measure its power consump-

Table 2: Key results. For each app, we provide app-description, size metrics, observed defects and energy-saving observed as result of applying the re-factoring suggested by our framework

Name(Version)	App Description	Apk Size (KB)	LoC	Energy Saving (%)	Re-factoring Description
Sensorium (1.1.12) [14]	Collect sensor data	1248	4001	21	Restricting use of sensors/GPS to key functionality. Adding resource release at exit.
UserHash (1.1 [17])	Location reporting service	171	837	15	Adding GPS release at exit.
Aripuca (1.3.4) [18]	Tracking app	660	8093	15	Adding GPS release at exit.
ShareMyPosition (1.0.11) [19]	Share your location	25	474	3	Replacing Full Wakelock with less-expensive counterparts.
DroidSat (2.47) [20]	Satellite Viewer	146	15007	4	Removing nesting of location resources. Replacing GPS uses with less-expensive counterparts.
iTLogger (1.0.0) [21]	Speed/heading information	553	4014	9	Replacing Full Wakelocks with less-expensive counterpart. Adding GPS, Sensor release at exit.
Heart Rate (1.0) [22]	Heart rate monitor	849	557	5	Replacing Screen Bright Wakelocks with less-expensive counterparts.
ChessClock (1.2.0) [13]	Touchable chess clock timer	336	725	14	Restricting use of Wakelock to key functionality. Replacing Full Wakelocks with less-expensive counterpart.
0xBenchmark (1.1.5) [23]	Mobile benchmark suite	1020	9739	29	Restricting use of Wakelock to key functionality. Adding resource release at exit.
Ham (1.5.7) [24]	Amateur radio tools	43	2224	6	Replacing GPS uses with less-expensive counterparts.

tion. The mobile device used in our experiments was Samsung S4, running an Android KitKat OS (version 4.4.2). To maintain consistency in power measurements across our experiments, we followed a few timing restriction (as shown in Figure 9(b)). For instance, while measuring the energy consumption of an app, the interval between two input-events (such as touches,taps,clicks) was 15 seconds. Additionally, an idle time (of 45 seconds), was observed just after the app had started or stopped execution. Finally, the screen time-out duration of the mobile device was set to 15 seconds. The inputs (to the app) were encoded as monkeyrunner scripts and were invoked from the Desktop PC. Our re-factoring framework and a power measurement utility, were run on a Desktop PC. The Desktop-PC was equipped with an Intel i7 processor, 8 GB main memory and Windows 7 OS.

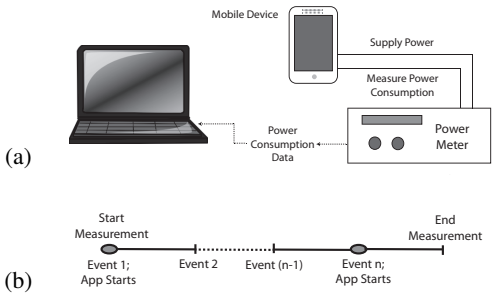


Figure 9: (a) Measurement setup (b) Timing parameters

4.2 Key Results

Of the ten apps studied in the evaluation, we found sub-optimal resource bindings in three apps, nested resource usage in one app, QoS trade-offs in six apps and resource leaks in five apps. Even though these apps are of considerable size (for instance, app *Sensorium* has $4,001$ LoC, apk size is $1,248$ KB), our framework was able to generate design expressions in less than a minute's time. This goes on to show that our technique can be scalably applied to real-world apps. It is worthwhile to know that the design expression generation time excludes the time required for EFG generation. This is because EFG is generated using a third-party tool Dynodroid and can be done off-line. It is also worthwhile to know that there is no termination criteria for the exploration algorithm in Dynodroid other than the number of events that it is allowed explored. As a result the time for EFG generation is heavily influenced by how many events the user wishes Dynodroid to explore.

Currently our framework can not only, produce the re-factored expression but also generate file/class names and method names, as well as lines numbers, where the re-factoring must be done. When we applied the re-factoring suggested by our framework, we observed a reduction in energy-consumption between 3 % to 29 %(*cf.* Table 2). One potential enhancement of our framework could be to implement this framework as an IDE plugin (such as an Eclipse Plugin), so that the framework continuously runs in the background, while monitoring and suggesting energy-efficient re-factorings while the app-developer is writing the code.

4.3 Case Study

Sensorium is a publicly available Android app which allows its user to collect sensor data such as network signal strength, location information, battery status, etc. We specifically choose this app out of the ten subject apps used in our evaluation because of its long and well maintained repository at GitHub [28]. This project was active for a period of approximately two years, in which duration it saw 214 commits. However, due to space restrictions, it would be impractical to discuss the design (& its re-factorings) for all of these 214 commits. Therefore, we choose only 6 important commits (referred to as commits a to f), by observing a plot of changes (both GUI, as well as, source code), as shown in Figure 10.

$$\text{Original Expression} \quad : \quad G_1G_2SE \tag{8}$$

$$\text{Re-factored Expression } \textit{(i)} \quad : \quad G_1G_2SG_1'G_2' \tag{9}$$

$$\text{Re-factored Expression } \textit{(ii)} \quad : \quad G_1SG_1' \tag{10}$$

During the earlier stages of the project (*cf.* commits $a - c$), the code and layout are changed heavily across consecutive commits (*cf.* Table 3) Whereas in the later commits (*i.e.* commits $d - f$), where the project is fairly mature and stable, the GUI layout and the design does not change substantially. During the evolution of the project, a number of commits had one or more re-factoring opportunities due to *sub-optimal binding* (due to sub-optimal sensor acquisitions), *nested usage* (such as nested location updates in commit b) and *resource leaks* (due to not releasing the sensor on app exit). These defects were successfully detected and re-factorings

suggested by our framework. For instance, commit b (design expression shown in expression 8, constituent symbols described in Table 3), could be re-factored for resource leaks, such as shown in expressions 9 and nested usage, such as shown in expression 10.

5. RELATED WORK

Increasingly large amount of research effort is being dedicated to understanding and resolving energy-inefficient behaviour in mobile apps. Here we shall discuss some of these works, specifically related to (i) Understanding energy-inefficient behaviour, (ii) Detecting energy-inefficient behaviour and (iii) Optimizing energy-consumption behaviour.

Understanding energy-inefficient behaviour: The first step in resolving energy-inefficient behaviour is to understand its characteristics. Recent works such as [4, 29–42] have presented some interesting insights towards understanding energy-inefficient behaviour in mobile apps. Profiling work such as [4] present insights such as the fact that I/O components (such as GPS, Wifi) and power management utilities (such as Android Wakelocks) are usually responsible for high energy consumption on mobile apps. Works such [29,32,33] have presented frameworks that use energy-models to estimate the energy consumption of an app, for a given workload. In particular, the works of [33] and [32] present techniques to map the energy-profile of an app (for a given input), to the app source-code. However, a key limitation of profiling-based techniques is that they heavily depend on test-inputs to generate the energy-profile. Manually, obtaining suitable test-inputs that can expose energy-inefficient behaviour is often non-trivial. In comparison, in our framework we use design expressions (more specifically, intersection between design expressions and defect expressions) to detect energy-inefficiencies in an app.

Detecting energy-inefficient behaviour: In the recent years, a number of works [5, 34, 43–48] have proposed dynamic, as well as static program analysis techniques for detecting energy-inefficient behaviour in mobile apps. Dynamic program analysis techniques such as [30], use symbolic execution to estimate the energy consumption for a given path in a program. Other works [43, 44], have used static program analysis techniques to detect the presence of resource leaks in apps. Test-generation techniques for mobile apps have been mostly applied to functional properties. However, a few works, such as [5, 45], exist that assist in energy-aware test-generation. Our recent work [49] presents a framework that uses energy-inefficient design patterns to debug and localize field failures in mobile-apps. In general, techniques described in this paragraph can assist in detecting energy-inefficiencies in an app post-development, however, such techniques do not provide support for energy-aware app re-factoring. In contrast, our technique is specifically designed to assist the app-developer by suggesting energy-efficient re-factorings, during the app-development stage.

Optimizing energy-consumption behaviour: A number of orthogonal approaches [50–53]have been presented over the recent year to optimize energy-efficiency of programs. For instance, [50] proposes the use of a new energy-aware programming language. Such languages, if used, can be instrumental in developing energy-efficient application, however, so far, such languages have not witnessed widespread use. Another group of work [51, 52] focuses on using energy-aware optimization. [51] in particular proposes the use of *approximate implementations* [51]. The key idea behind this work is to encode multiple, approximate implementations of a given (time-consuming) computation, such as loops. Since in mobile apps time-consuming computation may not necessarily imply energy intensive computation (because CPU may have a lesser power consumption than I/O components [5]), therefore, direct use of *approximate implementation* may not very beneficial for mobile-apps. However, the underlying philosophy of trade-offs between QoS and energy-efficiency is useful and therefore adapted to our technique as well. One of our previous articles [53], discusses the potential for energy-aware programming, however, it does not provide a framework necessary to conduct energy-aware re-factoring. Another preliminary work [54] proposes a re-factoring technique that uses compiler optimization to improve energy-efficiency of *Observer* and *Decorator* design patterns in object-oriented programs.

6. THREATS TO VALIDITY

A threat to the validity of our framework may arise due to the incompleteness of the EFG model. In our framework, a dynamic exploration technique that is used to create the EFG may not able to generate a complete UI model (EFG) for a given app. This may cause certain part of the app code to be unmodeled and hence unanalysed by our framework. It is worthwhile to mention alternative static analysis based techniques (such as the one used in [44]) that are based on parsing of XML-based UI files, may also be unable to generate a complete UI model for a given app because Android framework allows creation of dynamic UI screen programmatically. To the best our knowledge no existing work provides a technique for complete EFG generation. However, since the design expression generation part and the re-factoring part of our framework are loosely coupled with the EFG generation part, if any complete EFG generation technique is developed in future we can easily integrate it with our framework. Another threat to validity to this work may arise due to the choice of subject programs. Since we needed open source apps of our experiments we were restricted to Fdroid open-source app repository. Even though Fdroid is the biggest app repository of its kind, it is still small as compared to Google Play Store. This may have introduced some sampling bias [55] in our results.

7. DISCUSSION & CONCLUSION

In this paper, we present a technique to address the need for tools that can assist in energy-aware app development. Our technique uses a set of energy-efficiency guidelines to re-factor the design expression of an app. A design-expression is a regular-expression that represents the ordering of energy-intensive, resource usages and invocation of key functionalities (event-handlers) within the app. As result of using design-expressions, our re-factored technique is not limited by event-handler (class/method) boundaries. This not only increases the re-factoring opportunities but also makes our technique scalable. To demonstrate the efficacy of our technique we analysed a suite of open-source, apps with our technique. The resultant re-factoring when applied, reduced the energy-consumption of these apps between 3 % to 29 %. We also present a case study for one of our subject apps that captures its design evolution over a period of two-years and more than 200 commits. Our framework found re-factoring opportunities in a number of commits, that could have been implemented earlier on in the development stages, had the developer used an energy-aware re-factoring technique such as the one presented in this work.

Acknowledgement

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

Figure 10: Some commit from the 214 commits of the project *Sensorium*

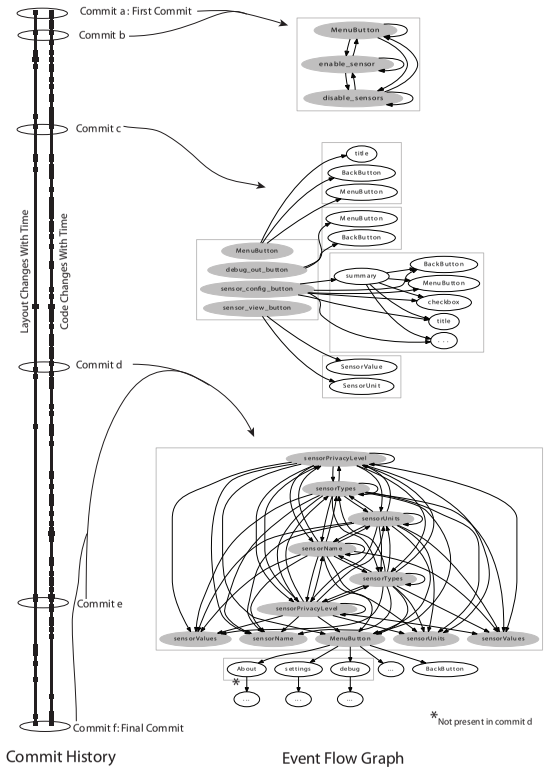


Table 3: Design expression and re-factorings for commits highlighted in Figure 10

Commit Hash (#)	Code Layout Changes	Design Expression	Re-factored Design Expression	Comments
73b0444 (a)	Initial commit. Event handler not attached to events.	SE	SE	n/a
9d141b7 (b)	Add more sensors (location)	G_1G_2SE	$G_1SG_1{}'E$	Resource Leaks; Nested Usage
94c58c3 (c)	Changed layout, removed files	$G_1G_2(((SM)\|SDD_1)\|SCC_1(G_1G_2\|G_1'G_2'))E\|G_1G_2SDD_1E\|G_1G_2SCC_1(G_1G_2\|G_1'G_2')T^*E$	$(((G_1SG_1'M)\|G_1SG_1'G_1DD_1G_1')\|G_1SG_1'CC_1)E\|G_1SG_1'G_1DD_1G_1'E\|G_1SG_1'CC_1T^*E$	Sub-optimal resource binding; Resource leaks; Nested Usage
e2aebfe (d)	Change layout of main screen	$G_1G_2S_1((((S\|SM^+)\|SM^+CC_1G_1G_2S_1)\|SM^+DD_1)\|SM^+C_1G_1'G_2'S_1{}'CC_1)E$	$((((G_1S_1SG_1'S_1'\|G_1S_1SG_1'S_1'M^+)\|G_1S_1SG_1'S_1'M^+CC_1)\|G_1S_1SG_1'S_1'M^+G_1S_1DD_1G_1'S_1')\|G_1S_1SG_1'S_1'M^+(CC_1))E$	Sub-optimal resource binding; Resource leaks; Nested Usage
fd643d7 (e)	Added more sensor (pressure)			
832aa14 (f)	Most recent commit			

E - Exit app
T - Summary
M - Menu button pressed
S - Main activity on screen
D - Debug activity
D_1 - Debug activity, (sub event 1)
C - Configuration activity
G_1 - Location acquire, first occurrence
G_2 - Location acquire, second occurrence
S_1 - Pressure sensor acquire, first occurrence
$G_1{}'$ - Release of resource G_1
$G_2{}'$ - Release of resource G_2
$S_1{}'$ - Release of resource S_1
C_1 - Configuration activity, (sub event 1)

8. REFERENCES

[1] Get started with publishing. http://developer.android.com/distribute/googleplay/start.html.

[2] Statista. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store.

[3] Monkeyrunner tool. http://developer.android.com/tools/help/MonkeyRunner.html.

[4] 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.

[5] A. Banerjee, L.K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE*, 2014.

[6] F-droid. https://f-droid.org/.

[7] 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.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

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

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

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

[12] Greenery. https://github.com/ferno/greenery.

[13] Chessclock. https://f-droid.org/repository/browse/?fdfilter=chessclock.

[14] Sensorium. https://f-droid.org/repository/browse/?fdfilter=Sensorium&fdid=at.univie.sensorium.

[15] App category. https://play.google.com/store/apps/category/APPLICATION?hl=en.

[16] Apache lucene core. https://lucene.apache.org/core/.

[17] Userhash. https://f-droid.org/repository/browse/?fdfilter=Userhash&fdid=com.threedlite.userhash.location.

[18] Aripuca. https://f-droid.org/repository/browse/?fdid=com.aripuca.tracker.

[19] Sharemylocation. https://f-droid.org/repository/browse/?fdfilter=sharemyposition&fdid=net.sylvek.sharemyposition.

[20] Droidsat. https://f-droid.org/repository/browse/?fdfilter=droidsat&fdid=com.mkf.droidsat.

[21] Itlogger. https://f-droid.org/repository/browse/?fdfilter=itlogger&fdid=de.tui.itlogger.

[22] Heart rate monitor. https://f-droid.org/repository/browse/?fdfilter=heartrate&fdid=com.vanderbie.heart_rate_monitor.

[23] 0xbenchmark. https://f-droid.org/repository/browse/?fdid=org.zeroxlab.zeroxbenchmark.

[24] Ham. https://f-droid.org/repository/browse/?fdfilter=Ham&fdid=com.smerty.ham.

[25] Soot. https://sable.github.io/soot/.

[26] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, 2013.

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

[28] Sensorium repository - github. https://github.com/fmetzger/android-sensorium.

[29] Lide Zhang, B. Tiwana, R.P. Dick, Zhiyun Qian, Z.M. Mao, Zhaoguang Wang, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, 2010.

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

[31] 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.

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

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

[34] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 2014.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in android applications. In *ASE*, 2013.

[45] Y. Liu, C. Xu, S.C. Cheung, and J. Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *Software Engineering, IEEE Transactions on*, 2014.

[46] Yepang Liu, Chang Xu, and S.C. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, 2013.

[47] 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.

[48] 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, 2012.

[49] Abhijeet Banerjee, Hai-Feng Guo, and Abhik Roychoudhury. Debugging energy-efficiency related field failures in mobile apps. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft, 16, 2016.

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

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

[52] 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.

[53] A. Banerjee and A. Roychoudhury. Energy-aware design patterns for mobile application development (invited talk). In *Proceedings of the 2Nd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2014, 2014.

[54] Adel Noureddine and Ajitha Rajan. Optimising energy consumption of design patterns. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, 2015.

[55] William Martin, Mark Harman, Yue Jia, Federica Sarro, and Yuanyuan Zhang. The app sampling problem for app store mining. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, 2015.