Using Automatic Refactoring to Improve Energy Efficiency of Android Apps

Luis Cruz[1] and Rui Abreu[2]

[1] University of Porto / HASLab, INESC TEC, luiscruz@fe.up.pt
[2] Instituto Superior Técnico, University of Lisbon / INESC-ID, rui@computer.org

Abstract. The ever-growing popularity of mobile phones has brought additional challenges to the software development lifecycle. Mobile applications (apps, for short) ought to provide the same set of features as conventional software, with limited resources: such as, limited processing capabilities, storage, screen and, not less important, power source. Although energy efficiency is a valuable requirement, developers often lack knowledge of best practices. In this paper, we study whether or not automatic refactoring can aid developers ship energy efficient apps. We leverage a tool, *Leafactor*, with five energy code smells that tend to go unnoticed. We use *Leafactor* to analyze code smells in 140 free and open source apps. As a result, we detected and fixed code smells in 45 apps, from which 40% have successfully merged our changes into the official repository.

1 Introduction

In the past decade, the advent of mobile devices has brought new challenges and paradigms to the existing computing models. One of the major challenges is the fact that mobile phones have a limited battery life. As a consequence, users need to frequently charge their devices to prevent their inoperability. Hence, energy efficiency is an important non-functional requirement in mobile software, with a valuable impact on usability.

A study in 2013 reported that 18% of apps have feedback from users that is related with energy consumption [25]. Other studies have nonetheless found that most developers lack the knowledge about best practices for energy efficiency in mobile applications (apps) [17, 23]. Hence, it is important to provide developers with actionable documentation and toolsets that aim to help deliver energy efficient apps.

Previously, we have identified code optimizations with significant impact on the energy consumption of Android apps [4]. However, certify that code is complying with these patterns is time-consuming and prone to errors. Thus, in this paper we study how automatic refactor can help develop code that follows energy best practices.

There are state-of-the-art tools that provide automatic refactoring for Android and Java apps (for instance, *AutoRefactor*[3], *Walkmod*[4], *Facebook pfff*[5], *Kadabra*[6]). Although these tools help developers creating better code, they do not feature energy related patterns for Android. Thus, we leverage five energy optimizations in an automatic refactoring tool, *Leafactor*, which is publicly available with an open source license. In addition, the toolset has the potential to serve as an educative tool to aid developers in understanding which practices can be used to improve energy efficiency.

On top of that, we analyze how Android developers are addressing energy optimizations and how an automatic refactoring tool would help ship more energy efficient mobile software. We have used the results of our tool to contribute to real Android app projects, validating the value of adopting an automatic refactoring tool in the development stack of mobile apps.

In a dataset of 140 free and open source software (FOSS) Android apps, we have found that a considerable part (32%) is released with energy inefficiencies. We have fixed 222 anti patterns in 45 apps, from which 18 have successfully merged our changes into the official branch. Results show that automatic refactoring tools can be very helpful to improve the energy footprint of apps.

This work is an extension of a previous two-page tooldemo that introduces *Leafactor* [5]. The remainder of this paper is organized as follows: Section 2 details energy refactorings and corresponding impact on energy consumption; in Section 3, we present the automatic refactor toolset that was implemented; Section 4 describes the experimental methodology used to validate our tool, followed by Sections 5 and 6 with results and discussion; in Section 7 we present the related work in this field; and finally Section 8 summarizes our findings and discusses future work.

2 Energy Refactorings

We use static code analysis and automatic refactoring to apply Android-specific optimizations of energy efficiency. In this section, we describe refactorings which are known to improve energy consumption of Android apps [4]. Each of them has an indication of the energy efficiency improvement (🍃) and the fix priority provided by the official *lint* documentation[7]. The priority reflects the severity of a pattern in terms of performance and is given in a scale of 1 to 10, with 10 being the most severe. The severity is not necessarily correlated with energy performance. In addition, we also provide examples where the refactorings are applied. All refactorings are in Java with the exception *ObsoleteLayoutParam* which is in XML — the markup language used in Android to define the user interface (UI).

[3] *AutoRefactor*: https://goo.gl/v5im9X (March 2, 2018).
[4] *Walkmod*: https://goo.gl/LmsUDX (March 2, 2018).
[5] *Facebook pfff*: https://goo.gl/NG1PTE (March 2, 2018).
[6] *Kadabra*: https://goo.gl/A5PsZf (March 2, 2018).
[7] *Lint* is a tool provided with the Android SDK which detects problems related with the structural quality of the code. Website: https://goo.gl/RA2EVC (March 2, 2018).

2.1 ViewHolder: View Holder Candidates

Energy efficiency improvement (🍃): 4.5%. Lint priority: |||||||||| 5/10.

This pattern is used to make a smoother scroll in *List Views*, with no lags. When in a *List View*, the system has to draw each item separately. To make this process more efficient, data from the previous drawn item should be reused. This technique decreases the number of calls to the method `findViewById()`, which is known for being a very inefficient method [14]. The following code snippet provides an example of how to apply *ViewHolder*.

```java
// ...
@Override
public View getView(final int position, View convertView, ViewGroup parent) {
  convertView = LayoutInflater.from(getContext()).inflate( ❶
    R.layout.subforsublist, parent, false
  );
  final TextView t = ((TextView) convertView.findViewById(R.id.name)); ❷
// ...
```

Optimized version:

```java
// ...
private static class ViewHolderItem { ❸
        private TextView t;
}

@Override
public View getView(final int position, View convertView, ViewGroup parent) {
  ViewHolderItem viewHolderItem;
  if (convertView == null) { ❹
    convertView = LayoutInflater.from(getContext()).inflate(
      R.layout.subforsublist, parent, false
    );
    viewHolderItem = new ViewHolderItem();
    viewHolderItem.t = ((TextView) convertView.findViewById(R.id.name));
    convertView.setTag(viewHolderItem);
  } else {
    viewHolderItem = (ViewHolderItem) convertView.getTag();
  }
  final TextView t = viewHolderItem.t; ❺
// ...
```

❶ In every iteration of the method `getView`, a new `LayoutInflater` object is instantiated, overwriting the method's parameter `convertView`.

❷ Each item in the list has a view to display text — a `TextView` object. This view is being fetched in every iteration, using the method `findViewById()`.

❸ A new class is created to cache common data between list items. It will be used to store the TextView object and prevent it from being fetched in every iteration.

❹ This block will run only in the first item of the list. Subsequent iterations will receive the `convertView` from parameters.

❺ It is no longer needed to call `findViewById()` to retrieve the `TextView` object.

One might argue that the version of the code after refactoring is considerably less intuitive. This is in fact true, which might be a reason for developers to ignore optimizations. However, regardless of whether this optimization should be taken care by the system, it is the recommended approach, as stated in the Android official documentation[8].

2.2 DrawAllocation: Allocations within drawing code

1.5%. Lint priority: |||||||||| 9/10.

Draw operations are very sensitive to performance. It is a bad practice allocating objects during such operations since it can create noticeable lags. The recommended fix is allocating objects upfront and reusing them for each drawing operation, as shown in the following example:

```java
public class DrawAllocationSampleTwo extends Button {
  public DrawAllocationSampleTwo(Context context) {
        super(context);
  }
  @Override
  protected void onDraw(android.graphics.Canvas canvas) {
      super.onDraw(canvas);
      Integer i = new Integer(5);❶
      // ...
      return;
  }
}
```

Optimized version:

```java
public class DrawAllocationSampleTwo extends Button {
  public DrawAllocationSampleTwo(Context context) {
        super(context);
  }
  Integer i = new Integer(5);❷
  @Override
  protected void onDraw(android.graphics.Canvas canvas) {
    super.onDraw(canvas);
    // ...
    return;
  }
}
```

❶ A new instance of `Integer` is created in every execution of `onDraw`.

❷ The allocation of the instance of `Integer` is removed from the drawing operation and is now executed only once during the app execution.

2.3 WakeLock: Incorrect wake lock usage

1.5%. Lint priority: |||||||||| 9/10.

Wake locks are mechanisms to control the power state of a mobile device. This can be used to prevent the screen or the CPU from entering a sleep state. If an application fails to release a wake lock, or uses it without being strictly necessary, it can drain the battery of the device.

The following example shows an Activity that uses a wake lock:

[8] *ViewHolder* explanation in the official documentation: https://goo.gl/tgy7xL visited in March 2, 2018.

```
extends Activity { private WakeLock wl;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PowerManager pm = (PowerManager) this.getSystemService(
      Context.POWER_SERVICE
    );
    wl = pm.newWakeLock(
      PowerManager.SCREEN_DIM_WAKE_LOCK | PowerManager.ON_AFTER_RELEASE,
      "WakeLockSample"
    );
    wl.acquire();❶
  }
}
```

❶ Using the method `acquire()` the app asks the device to stay on. Until further instruction, the device will be deprived of sleep.

Since no instruction is stopping this behavior, the device will not be able to enter a sleep mode. Although in exceptional cases this might be intentional, it should be fixed to prevent battery drain.

The recommended fix is to override the method `onPause()` in the activity:

```
//...
@Override protected void onPause(){
  super.onPause();
  if (wl != null && !wl.isHeld()) {
    wl.release();
  }
}
//...
```

With this solution, the lock is released before the app switches to background.

2.4 Recycle: Missing `recycle()` calls

0.7%. Lint priority: |||||||||| 7/10.

There are collections such as `TypedArray` that are implemented using singleton resources. Hence, they should be released so that calls to different `TypedArray` objects can efficiently use these same resources. The same applies to other classes (e.g., database cursors, motion events, etc.).

The following snippet shows an object of `TypedArray` that is not being recycled after use:

```
public void wrong1(AttributeSet attrs, int defStyle) {
  final TypedArray a = getContext().obtainStyledAttributes(
    attrs, new int[] { 0 }, defStyle, 0
  );
  String example = a.getString(0);
}
```

Solution:

```
public void wrong1(AttributeSet attrs, int defStyle) {
  final TypedArray a = getContext().obtainStyledAttributes(
    attrs, new int[] { 0 }, defStyle, 0
  );
  String example = a.getString(0);
  if (a != null) {
    a.recycle();❶
  }
}
```

❶ Calling the method `recycle()` when the object is no longer needed, fixes the issue. The call is encapsulated in a conditional block for safety reasons.

2.5 ObsoleteLayoutParam (OLP): Obsolete layout params

0.7%. Lint priority: |||||||||| 6/10.

During development, UI views might be refactored several times. In this process, some parameters might be left unchanged even when they have no effect in the view. This causes useless attribute processing at runtime. As example, consider the following code snippet (XML):

```
<LinearLayout>
  <TextView android:id="@+id/name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"> /* DeleteMe */ ❶
  </TextView>
</LinearLayout>
```

❶ The property `android:layout_alignParentBottom` is used for views inside a `RelativeLayout` to align the bottom edge of a view (i.e., the `TextView`, in this example) with the bottom edge of the `RelativeLayout`. On contrary, `LinearLayout` is not compatible with this property, having no effect in this example. It is safe to remove the property from the specification.

3 Automatic Refactoring Tool

In the scope of our study, we developed a tool to statically analyze and transform code, implementing Android-specific energy-efficiency optimizations — *Leafactor*. The toolset receives a single file, a package, or a whole Android project as input and looks for eligible files, i.e., Java or XML source files. It automatically analyzes those files and generates a new compilable and optimized version.

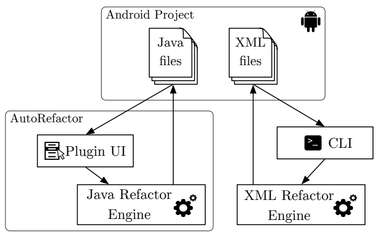
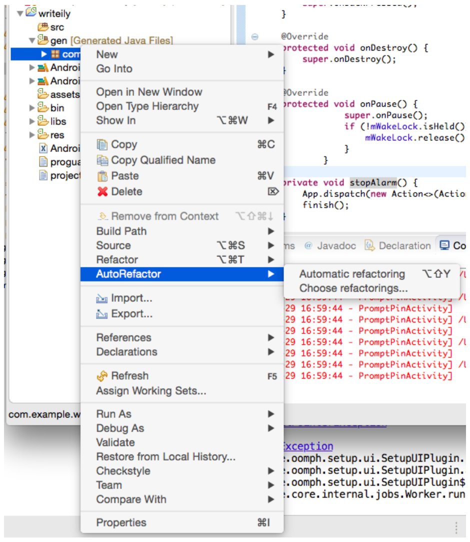


Fig. 1. Architecture diagram of the automatic refactoring toolset.

The architecture of *Leafactor* is depicted in Figure 1. There are two separate engines: one to handle Java files and another to handle XML files. The refactoring engine for Java is implemented as part of the open-source project *AutoRefactor* — an *Eclipse* plugin to automatically refactor Java code bases.

AutoRefactor provides a comprehensive set of common code cleanups to help deliver "smaller, more maintainable and more expressive code bases"[9]. Eclipse Marketplace[10] reported 2884 successful installs of *AutoRefactor*. A common use case is presented in the screenshot of Figure 2. Under the hood, *AutoRefactor* integrates a handy and concise API to manipulate Java *Abstract Syntax Trees* (ASTs). We contributed to the project by implementing the Java refactorings mentioned in Section 2.



```
}

@Override
protected void onDestroy() {
    super.onDestroy();
}

@Override
protected void onPause() {
        super.onPause();
        if (!mWakeLock.isHeld()
            mWakeLock.release()
        }
    }

private void stopAlarm() {
    App.dispatch(new Action<>(Actio
    finish();
}
```

```
29 16:59:44 - PromptPinActivity]
29 16:59:44 - PromptPinActivity] /l
29 16:59:44 - PromptPinActivity]
29 16:59:44 - PromptPinActivity] /l
29 16:59:44 - PromptPinActivity]
29 16:59:44 - PromptPinActivity] /l
29 16:59:44 - PromptPinActivity]
```

```
Exception
.oomph.setup.ui.SetupUIPlugin.
.oomph.setup.ui.SetupUIPlugin.
.oomph.setup.ui.SetupUIPlugin$
.core.internal.jobs.Worker.run
```

Fig. 2. Developers can apply refactoring by selecting the "Automatic refactoring" option or by using the key combination $\boxed{\text{⌥}}+\boxed{\text{⇧}}+\boxed{\text{Y}}$.

Since XML refactorings are not supported by *AutoRefactor*, a separate refactoring engine was developed and integrated in *Leafactor*. As detailed in the previous section, only a single XML refactoring is offered — *ObsoleteLayoutParam*.

4 Empirical evaluation

We designed an experiment with the following goals:

– Study the benefits of using an automatic refactoring tool within the Android development community.
– Study how FOSS Android apps are adopting energy efficiency optimizations.
– Improve energy efficiency of FOSS Android apps.

[9] As described in the official website, visited in March 2, 2018: https://goo.gl/v5im9X
[10] *Eclipse Marketplace* is an interface for browsing and installing plugins for the Java IDE Eclipse: https://goo.gl/QkTcWm visited in March 2, 2018.

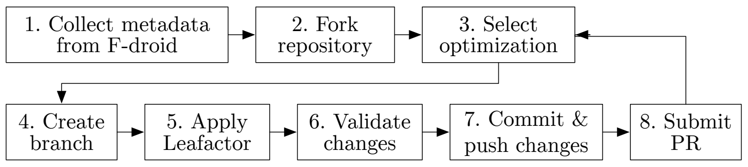


Fig. 3. Experiment's procedure for a single app.

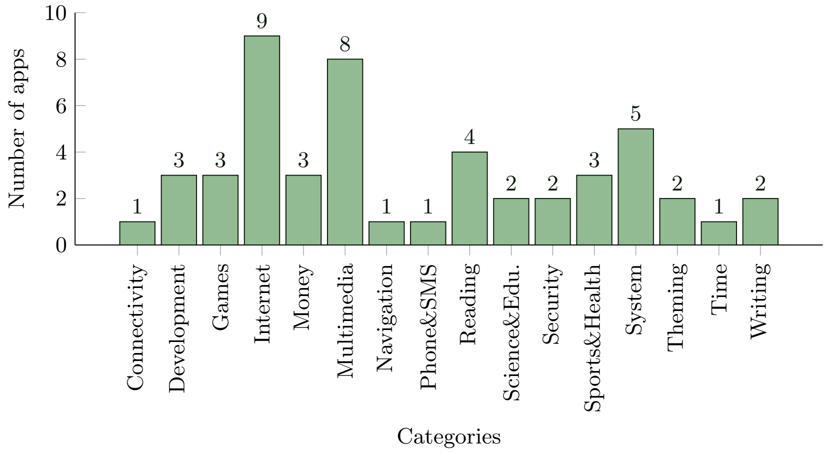


Fig. 4. Number of apps per category in the dataset.

We adopted the procedure explained in Figure 3. Starting with step *1*, we collect data from the *F-droid* app store[11] — a catalog for free and open-source software (FOSS) applications for the Android platform. For each app we collect the git repository location which is used in step *2* to fork the repository and prepare it for a potential contribution in the app's official project. Following, in step *3* we select one refactoring to be applied and consequently initiate a process that is repeated for all refactorings (steps *4–8*): the project is analyzed and, if any transformation is applied, a new *Pull Request* (PR) is submitted to be considered by the project's integrator. Since we wanted to engage the community and get feedback about the refactorings, we manually created each PR with a personalized message, including a brief explanation of commited code changes.

We analyze 140 free and open-source Android apps collected from *F-droid*[12]. Apps are selected by date of publish (i.e., it was given priority to newly released apps), considering exclusively Java projects (e.g., *Kotlin* projects are filtered out) with a *Github* repository. We select only one git service for the sake of simplicity. Apps in the dataset are spread in 17 different categories, as depicted in Figure 4.

[11] F-droid repository is available at https://goo.gl/cj8fC8 visited in March 2, 2018.
[12] Data was collected in Nov 27, 2016 and it is available here: https://goo.gl/CrmUEz

The largest project in terms of Java files is *TinyTravelTracker* (1878), while *NewsBlue* is the largest in terms of XML files (2109). Table 1 presents descriptive statistics for the source code and repository of the projects in the dataset. In total we analyzed 6.79GB of Android projects in 4.5 hours, totaling 15308 Java files and 15103 XML files.

Table 1. Descriptive statistics of projects in the dataset.

	Java Files	XML Files	Github Forks	Github Stars	Contributors
Mean	103	102	65	179	15
Min	0	4	0	0	1
25%	13	23	3.75	7.75	2
Median	38	48	9	24	3
75%	106	97	39	111	10
Max	1678	2109	1483	4488	323
Total	15308	15103	9547	26484	2162

5 Results

Our experiment yielded a total of 222 refactorings, which were submitted to the original repositories as PRs. Multiple refactorings of the same type were grouped in a single PR to avoid creating too many PRs for a single app. It resulted in 59 PRs spread across 45 apps. This is a demanding process, since each project has different contributing guidelines. Nevertheless, by the time of writing, 18 apps had successfully merged our contributions for deployment.

Table 2. Summary of refactoring results

Refactoring	ViewHolder	DrawAllocation	Wakelock	Recycle	OLP*	Any
Total Refactorings	7	0	1	58	156	222
Total Projects	5	0	1	23	30	45
Percentage of Projects	4%	0%	1%	16%	21%	32%
Incidence per Project	1.4×	-	1.0×	2.5×	5.2×	4.8×

*OLP — ObsoleteLayoutParam

Table 2 presents the results for each refactoring. It shows the total number of applied refactorings, the total number of projects that were affected, the percentage of affected projects, and the average number of refactorings per affected project.

ObsoleteLayoutParam was the most frequent pattern. It was applied 156 times in a total of 30 projects out of the 140 in our dataset (21%). In average, each affected project had 5 occurrences of this pattern. *Recycle* comes next, occurring in 16% of projects (58 refactorings). *DrawAllocation* and *Wakelock* only showed marginal impact. In addition, Table 2 also presents the combined results for the occurrence of any type of refactoring (*Any*). In addition, Figure 5

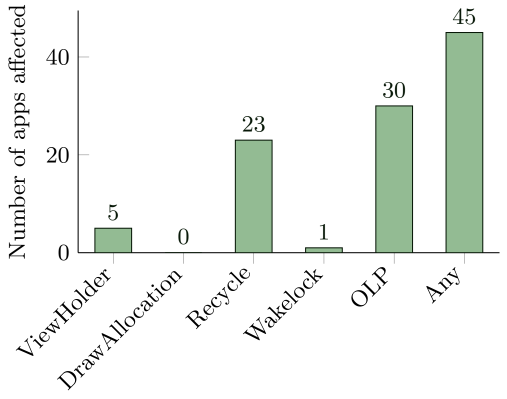


Fig. 5. Number of apps affected per refactoring.

presents a plot bar summarizing the number of projects affected amongst all the studied refactorings.

For reproducibility and a clarity of results, all the data collected in this study is publicly available[13]. All the PRs are public and can be accessed through the official repositories of the apps. As example, the PR for the refactoring *ViewHolder* performed in the app *Slide*[14] can be found in the *Github* project `ccrama/Slide` with reference `#2346`[15].

6 Discussion

Results show that an automatic refactoring tool can help developers ship more energy efficient apps. A considerable part of apps in this study (32%) had at least one energy inefficiency. Since these inefficiencies are only visible after long periods of app activity they can easily go unnoticed. From the feedback developers provided in the PRs, we have noticed that developers are open to recommendations from an automated tool. Only in a few exceptions, developers expressed being unhappy with our contributions. Most developers were curious about the refactorings and they recognized being unaware of their impact on energy efficiency. This is consistent with previous work [17, 23].

In a few cases, code smells were found in code that does not affect the energy consumption of the app itself (e.g., test code). In those cases, our PRs were not merged. Nevertheless, we recommend consistently complying with these optimizations in all types of code since new developers often use tests to help understand how to contribute to a project.

The code smell related with *ObsoleteLayoutParam* was found in a considerable fraction of projects (21%). This relates with the fact that app views are

[13] Spreadsheet with all experimental results: https://goo.gl/CrmUEz.
[14] *Slide*'s website: https://goo.gl/8HuJ6g visited in March 2, 2018.
[15] PR of the *ViewHolder* of app *Slide*: https://goo.gl/P2gFBx visited in March 2, 2018.

often created in an iterative process with several rounds of trial and error. Since some parameters have no effect under very specific contexts, useless lines of specification can go unnoticed by developers.

Recycle is frequent too, being observed in 16% of projects. This pattern is found in Android API objects that can be found in most projects (e.g., database cursors). Although a clean fix is to use the Java *try-with-resources* statement[16], it requires version 19 or earlier of Android SDK (introduced with Android 4.4 Kitkat). However, developers resort to a more verbose approach for backwards compatibility which requires explicitly closing resources, hence prone to mistakes.

Our *DrawAllocation* checker did not yield any result. It was expected that developers were already aware of *DrawAllocation*. Still, we were able to manually spot allocations that were happening inside a drawing routine. Nevertheless, those allocations are using dynamic values to initialize the object. In our implementation, we scope only allocations that will not change between iterations. Covering those missed cases would require updating the allocated object in every iteration. While spotting these cases is relatively easy, refactoring would require better knowledge of the class that is being instantiated. Similarly, *WakeLocks* are very complex mechanisms and fixing all misuses still requires further work.

In the case of *ViewHolder*, although it only impacted 4% of the projects, we believe it has to do with the fact that 1) some developers already know this pattern due to its performance impact, and 2) many projects do not implement dynamic list views. *ViewHolder* is the most complex pattern we have in terms of lines of code (LOC) — a simple case can require changes in roughly 35 LOC. Although changes are easily understandable by developers, writing code that complies with *ViewHolder* pattern is not intuitive.

A positive outcome of our experimentation was that we were able to improve energy efficiency in the official release of 18 Android apps.

7 Related Work

Energy efficiency of mobile apps is being addressed with many different approaches. Some works opt by simplifying the process of measuring energy consumption of mobile apps [26, 18, 19, 9, 6, 3]. Alternatively, other works study the energy footprint of software design choices and code patterns that will prevent developers from creating code with poor energy efficiency [13, 11, 12, 15, 16, 20].

Automatic detection of anti-patterns for Android has been studied before. Fixing code smells in Android has shown gains up to 5% in energy efficiency [4]. Code was manually refactored in six real apps and energy consumption was measured using an hardware based power monitor. Our work extends this research by providing automatic refactoring to the resulting energy code smells.

The frequency of anti-patterns in Android apps was studied in previous work [10]. Patterns were automatically detected in 15 apps using the tool *Paprika*

[16] Documentation about the Java *try-with-resources* statement: https://goo.gl/5TmSkc visited in March 2, 2018.

which was developed to perform static analysis in the bytecode of apps. Although Paprika provides valuable feedback on how to fix their code, developers need to manually apply the refactorings. Our study differs by focusing on energy related anti-patterns and by applying automatic refactoring to resolve potential issues.

Previous work has also studied the importance of providing a catalogue of bad smells that negatively influence specific quality requirements, such as energy efficiency [22, 21]. Although the authors motivate the importance of using automatic refactoring, their approach lacks an extensive implementation of their catalogue. In this work, we validate our refactorings by applying *Leafactor* in a large dataset of real Android apps. Moreover, we assess how automatic refactoring tools for energy can positively impact the Android FOSS community.

Other works have detected energy related code smells by analyzing source code as *TGraphs* [8, 7]. Eight different code smell detectors were implemented and validated with a navigation app. Fixing the code with automatic refactoring was discussed but not implemented. In addition, although studied code smells are likely to have an impact on energy consumption, no evidence was presented.

Previous work have used the event flow graph of the app to optimize resource usage (e.g., GPS, Bluetooth) [2]. Results show significant gains in energy efficiency. Nevertheless, although this process provides details on how to fix the code, it is not fully automated yet.

Other works have studied and applied automatic refactorings in Android applications [23, 24]. However, these optimizations were not mobile specific.

Besides refactoring source code, other works have focused in studying the impact of UI design decisions on energy consumption [15]. Agolli, T., et al. have proposed a methodology that suggests changes in the UI colors of apps. The new UI colors, despite being different, are almost imperceptible by users and lead to savings in the energy consumption of mobile phones' displays [1]. In our work, we strictly focus on changes that do not change the appearance of the app.

8 Conclusion

Our work shows the potential of using automatic refactoring tools to improve energy efficiency of mobile applications. We have analyzed 140 FOSS Android apps and as an outcome we have fixed 222 energy related anti-patterns. In total, we improved the energy footprint of 45 apps.

As future work, we plan to study and support more energy refactorings. In addition, it would be interesting to integrate automatic refactoring in a continuous integration context. The integration would require two distinct steps: one for the detection and another for the code refactoring which would only be applied upon a granting action by a developer. One could also use this idea with an educational purpose. A detailed explanation of the code transformation along with its impact on energy efficiency could be provided whenever a developer pushes new changes to the repository.

Acknowledgments

This work is financed by the ERDF European Regional Development Fund through the Operational Program for Competitiveness and Internationalization - COMPETE 2020 Program and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718. Luis Cruz is sponsored by an FCT scholarship grant number PD/BD/52237/2013.

References

1. Agolli, T., Pollock, L., Clause, J.: Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. pp. 30–34. IEEE Press (2017)
2. Banerjee, A., Roychoudhury, A.: Automated re-factoring of android apps to enhance energy-efficiency. In: Proceedings of the International Workshop on Mobile Software Engineering and Systems. pp. 139–150. ACM (2016)
3. Couto, M., Carção, T., Cunha, J., Fernandes, J.P., Saraiva, J.: Detecting anomalous energy consumption in android applications. In: Brazilian Symposium on Programming Languages. pp. 77–91. Springer (2014)
4. Cruz, L., Abreu, R.: Performance-based guidelines for energy efficient mobile applications. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. pp. 46–57. IEEE Press (2017)
5. Cruz, L., Abreu, R., Rouvignac, J.N.: Leafactor: Improving energy efficiency of android apps via automatic refactoring. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. pp. 205–206. MOBILESoft '17, IEEE Press (2017)
6. Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., De Lucia, A.: Petra: a software-based tool for estimating the energy profile of android applications. In: Proceedings of the 39th International Conference on Software Engineering Companion. pp. 3–6. IEEE Press (2017)
7. Ebert, J., Riediger, V., Winter, A.: Graph technology in reverse engineering–the tgraph approach. In: Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics. Citeseer (2008)
8. Gottschalk, M., Josefiok, M., Jelschen, J., Winter, A.: Removing energy code smells with reengineering services. GI-Jahrestagung 208, 441–455 (2012)
9. Hao, S., Li, D., Halfond, W.G., Govindan, R.: Estimating mobile application energy consumption using program analysis. In: Software Engineering (ICSE), 2013 35th International Conference on. pp. 92–101. IEEE (2013)
10. Hecht, G., Rouvoy, R., Moha, N., Duchien, L.: Detecting antipatterns in android apps. In: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems. pp. 148–149. IEEE Press (2015)
11. Li, D., Halfond, W.G.: An investigation into energy-saving programming practices for android smartphone app development. In: Proceedings of the 3rd International Workshop on Green and Sustainable Software. pp. 46–53. ACM (2014)
12. Li, D., Halfond, W.G.: Optimizing energy of http requests in android applications. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile. pp. 25–28. ACM (2015)

13. Li, D., Hao, S., Gui, J., Halfond, W.G.: An empirical study of the energy consumption of android applications. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on. pp. 121–130. IEEE (2014)
14. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., Poshyvanyk, D.: Mining energy-greedy api usage patterns in android apps: an empirical study. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 2–11. ACM (2014)
15. Linares-Vásquez, M., Bernal-Cárdenas, C., Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D.: Gemma: multi-objective optimization of energy consumption of guis in android apps. In: Proceedings of the 39th International Conference on Software Engineering Companion. pp. 11–14. IEEE Press (2017)
16. Malavolta, I., Procaccianti, G., Noorland, P., Vukmirović, P.: Assessing the impact of service workers on the energy efficiency of progressive web apps. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. pp. 35–45. IEEE Press (2017)
17. Pang, C., Hindle, A., Adams, B., Hassan, A.E.: What do programmers know about the energy consumption of software? PeerJ PrePrints 3, e886v1 (2015)
18. Pathak, A., Hu, Y.C., Zhang, M.: 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. pp. 29–42. ACM (2012)
19. Pathak, A., Hu, Y.C., Zhang, M., Bahl, P., Wang, Y.M.: Fine-grained power modeling for smartphones using system call tracing. In: Proceedings of the sixth conference on Computer systems. pp. 153–168. ACM (2011)
20. Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J.P., Saraiva, J.: Helping programmers improve the energy efficiency of source code. In: Proceedings of the 39th International Conference on Software Engineering Companion. pp. 238–240. IEEE Press (2017)
21. Reimann, J., Aβmann, U.: Quality-aware refactoring for early detection and resolution of energy deficiencies. In: Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing. pp. 321–326. IEEE Computer Society (2013)
22. Reimann, J., Brylski, M., Aßmann, U.: A tool-supported quality smell catalogue for android developers. In: Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM. vol. 2014 (2014)
23. Sahin, C., Pollock, L., Clause, J.: How do code refactorings affect energy usage? In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. p. 36. ACM (2014)
24. Sahin, C., Pollock, L., Clause, J.: From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. Journal of Systems and Software 117, 307–316 (2016)
25. Wilke, C., Richly, S., Gotz, S., Piechnick, C., Aßmann, U.: Energy consumption and efficiency in mobile applications: A user feedback study. In: Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing. pp. 134–141. IEEE (2013)
26. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R.P., Mao, Z.M., Yang, L.: 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. pp. 105–114. ACM (2010)