


# Improving Energy Efficiency Through Automatic Refactoring

Luis Cruz  [ INESC-ID, University of Porto | [luisacruz@fe.up.pt](mailto:luisacruz@fe.up.pt) ]

Rui Abreu  [ INESC-ID, IST, University of Lisbon | [rui@computer.org](mailto:rui@computer.org) ]

## Abstract

The ever-growing popularity of mobile phones has brought additional challenges to the software development lifecycle. Mobile applications 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 propose a tool to improve the energy efficiency of Android applications using automatic refactoring — *Leafactor*. The tool features five energy code smells that tend to go unnoticed. In addition, to evaluate the effectiveness of our approach, we run an experiment over a dataset of 140 free and open source apps. As a result, we detected and fixed code smells in 45 Android apps, from which 40% have successfully merged our changes into the official repository.

**Keywords:** *Automatic Refactoring, Mobile Computing, Energy Efficiency, Software Engineering*

## 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 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 to energy consumption (Wilke et al., 2013). Other studies have nonetheless found that most developers lack the knowledge about best practices for energy efficiency in mobile applications (apps) (Pang et al., 2015; Sahin et al., 2014). Hence, it is important to provide developers with actionable documentation and toolsets that aim to help deliver energy efficient apps.

Previously, we have identified five code smells with significant impact on the energy consumption of Android apps (Cruz and Abreu, 2017) — we refer to them as *energy-related smells*. We used a hardware-based approach to assess the energy efficiency improvement of fixing eight performance-based code smells described in the official Android documentation. The impact on energy efficiency was evaluated by manually refactoring the codebases of five open-source Android applications. The energy consumption was measured for every pair of versions: before and after the refactoring. The measurements were performed by mimicking real use-case scenarios while collecting power data with the single-board computer ODROID<sup>1</sup>, which features power sensors for energy measurements. From those eight refactorings, five were found to yield a significant improvement in the energy consumption of mobile apps. However, certify that code is complying with these optimizations 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*<sup>2</sup>, *Walkmod*<sup>3</sup>, *Facebook pfff*<sup>4</sup>, *Kadabra*<sup>5</sup>). Although these tools help developers creating better code, they do not feature energy-related refactorings 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-related smells 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 energy-related smells 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 paper is an extension of our previous work, in which we introduced the automatic refactoring tool *Leafactor* (Cruz et al., 2017; Cruz and Abreu, 2018) for the first time. We provide a self-contained report of our work on improving energy efficiency of mobile apps via automatic refactorings, by adding details of the architecture of the toolset and the available set of refactorings. Moreover, we make a more comprehensive description of the dataset used in the empirical study, including complexity metrics. Combined, our work makes the following contributions:

<sup>2</sup>*AutoRefactor*: <http://autorefactor.org> (August 17, 2019).

<sup>3</sup>*Walkmod*: <http://walkmod.com> (August 17, 2019).

<sup>4</sup>*Facebook pfff*: <https://github.com/facebookarchive/pfff> (August 17, 2019).

<sup>5</sup>*Kadabra*: <http://specs.fe.up.pt/tools/kadabra/> (August 17, 2019).

<sup>1</sup>ODROID is a single-board computer that runs Android and is used for mobile application development and IoT applications.


- An automated refactoring tool, *Leafactor*, to improve energy efficiency of Android applications.
- An empirical study of the prevalence of five energy-related code smells in FOSS Android applications.
- The submission of 59 pull requests to the official code bases of 45 FOSS Android applications, comprehending 222 energy efficiency refactorings.

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 the energy consumption of Android apps. Each of them has an indication of the energy efficiency improvement (🍃), as assessed in previous work (Cruz and Abreu, 2017), and the fix priority provided by the official *lint* documentation<sup>6</sup>. The priority reflects the impact of the refactoring in terms of performance and is given on a scale of 1 to 10, with 10 being the most effective. 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).

### 2.1 ViewHolder: Add View Holder to scrolling lists

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

This refactoring 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 (Linares-Vásquez et al., 2014). The following code snippet provides an example of how to apply *ViewHolder*.

```
// ...
@Override
public View getView(final int position, View convertView,
    ViewGroup parent) {
    convertView = LayoutInflater.from(getContext()).inflate
        (
            ❶
            R.layout.subforsublist, parent, false
        );
}
```

<sup>6</sup>*Lint* is a tool provided with the Android SDK which detects problems related with the structural quality of the code. Website: <https://developer.android.com/studio/write/lint> (August 17, 2019).

```
final TextView t = ((TextView) convertView.findViewById
    (R.id.name)); ❷
// ...
```

Optimized version:

```
// ...
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 addressed by the system, it is the recommended approach, as stated in the Android official documentation<sup>7</sup>. See more on this discussion in Section 6.

### 2.2 DrawAllocation: Remove 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:

```
public class DrawAllocationSampleTwo extends Button {
    public DrawAllocationSampleTwo(Context context) {
        super(context);
    }
    @Override
    protected void onDraw(android.graphics.Canvas canvas) {
```

<sup>7</sup>*ViewHolder* explanation in the official documentation: <https://developer.android.com/guide/topics/ui/layout/recyclerview> visited in August 17, 2019.

```

    super.onDraw(canvas);
    Integer i = new Integer(5);❶
    // ...
    return;
}
}

```

Optimized version:

```

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: Fix incorrect wakelock usage

🍃 1.5%. Lint priority: ■■■■■■■■■ □ 9/10.

Wakelocks 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 wakelock 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:

```

extends Activity { private WakeLock wl;

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

    PackageManager pm = (PackageManager) this.
        getSystemService(
            Context.POWER_SERVICE
        );
    wl = pm.newWakeLock(
        PackageManager.SCREEN_DIM_WAKE_LOCK | PackageManager.
            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: Fix 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.

Besides `TypedArray` instances, this refactoring is also applied to instances of following classes: `Cursor`, `VelocityTracker`, `MotionEvent`, `Parcel`, and `ContentProviderClient`.

## 2.5 ObsoleteLayoutParam (OLP): Remove obsolete layout parameters

🍃 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 is a code smell that needs to be fixed since it causes useless attribute processing at runtime. The refactoring is applied by removing the obsolete parameters from the UI specification. As an example, consider the following code snippet (XML):

```

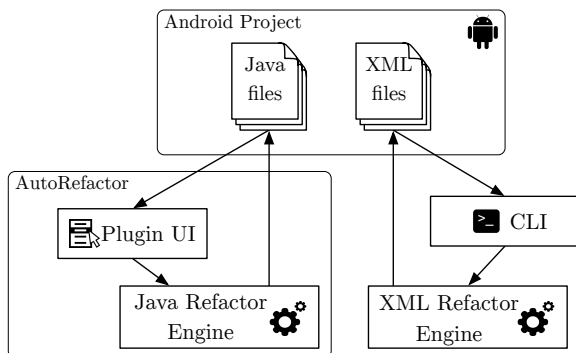
<LinearLayout>
    <TextView android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"> /* DeleteMe
        */ ❶
    </TextView>
</LinearLayout>

```

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

**Table 1.** Layout-related parameters that only have a visual effect when defined inside specific layouts.

Layout Parameter	Allowed Parent Layout
layout_x	AbsoluteLayout
layout_y	AbsoluteLayout
layout_weight	LinearLayout, ActionMenuView, ListRowHoverCardView, ListRowView, NumberPicker, RadioGroup, SearchView, TabWidget, TableLayout, TableRow, TextInputLayout, ZoomControls
layout_column	GridLayout, TableLayout, TableRow
layout_columnSpan	GridLayout
layout_row	GridLayout
layout_rowSpan	GridLayout
layout_alignLeft	RelativeLayout
layout_alignStart	RelativeLayout
layout_alignRight	RelativeLayout
layout_alignEnd	RelativeLayout
layout_alignTop	RelativeLayout
layout_alignBottom	RelativeLayout
layout_alignParentTop	RelativeLayout
layout_alignParentBottom	RelativeLayout
layout_alignParentLeft	RelativeLayout
layout_alignParentStart	RelativeLayout
layout_alignParentRight	RelativeLayout
layout_alignParentEnd	RelativeLayout
layout_alignWithParentMissing	RelativeLayout
layout_alignBaseline	RelativeLayout
layout_centerInParent	RelativeLayout
layout_centerVertical	RelativeLayout
layout_centerHorizontal	RelativeLayout
layout_toRightOf	RelativeLayout
layout_toEndOf	RelativeLayout
layout_toLeftOf	RelativeLayout
layout_toStartOf	RelativeLayout
layout_below	RelativeLayout
layout_above	RelativeLayout

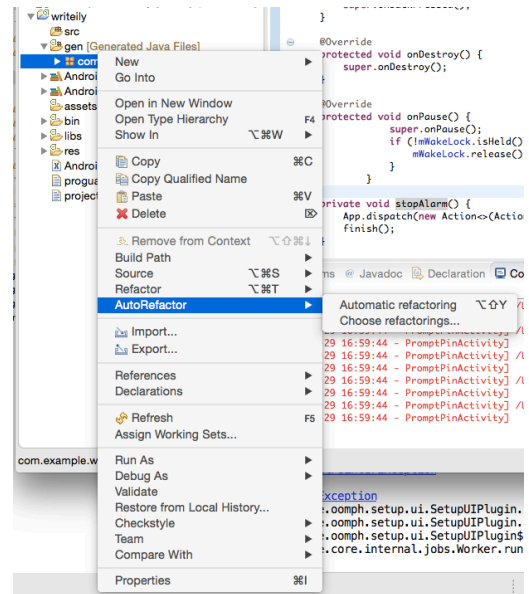
**Figure 1.** Architecture diagram of the automatic refactoring toolset.

from the specification. In Table 1, we detail all the cases featured in *Leafactor*.

### 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 refactorings — *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.

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

**Figure 2.** Developers can apply refactorings by selecting the “Automatic refactoring” option or by using the key combination  $\text{Ctrl} + \text{Alt} + \text{Y}$ .

bases.

#### 3.1 AutoRefactor

*AutoRefactor* is an Eclipse plugin that delivers automatic refactoring in Java codebases. It is created as a complement to existing static analyzers such as *SonarQube*, *FindBugs*, *CheckStyle* and *PMD*. Although they provide insightful warnings to developers, they do little in helping developers fixing all the issues lying in legacy codebases.

It provides a comprehensive set of 103 common code cleanups to help deliver “smaller, more maintainable and more expressive code bases”<sup>8</sup>. The list goes from simple rules, such as enforcing the use of the method `isEmpty()` to check whether a collection is empty, instead of checking its size (rule *IsEmptyRatherThanSize*), to more complex ones, such as *SetRatherThanList* choosing a more adequate collection type for specific use cases. In addition, *AutoRefactor* also supports cleanups for code comments, such as removing auto-generated or empty Javadocs from the codebase (named by *AutoRefactor* as rule *Comments*).

Eclipse Marketplace<sup>9</sup> reported 4459 successful installs of *AutoRefactor*.

A common use case is presented in the screenshot of Figure 2. Developers can apply refactorings in single files, packages, or entire projects.

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.

#### 3.2 XML refactorings

Since XML refactorings are not supported by *AutoRefactor*, a separate refactoring engine was developed and integrated

<sup>8</sup>As described in the official website, visited in August 17, 2019: <http://autorefactor.org>

<sup>9</sup>*Eclipse Marketplace* is an interface for browsing and installing plugins for the Java IDE Eclipse: <https://marketplace.eclipse.org> visited in August 17, 2019.



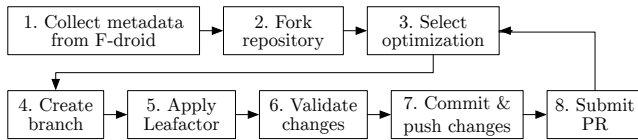


Figure 3. Experiment's procedure for a single app.

into *Leafactor*. The engine features a command line interface, that can be integrated with continuous integration environments. Optionally, the tool can be set to simply flag warnings, without performing any refactoring transformation. 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.

We adopted the procedure explained in Figure 3. Starting with step 1, we collected data from the *F-droid* app store<sup>10</sup> — a catalog for free and open-source software (FOSS) applications for the Android platform. For each mobile application, we collected the git repository location which was used in step 2 to fork the repository and prepare it for a potential contribution to the project's official code repository. Following, in step 3 we selected one refactoring to be applied and consequently initiate a process that was repeated for all refactorings (steps 4–8): the project was analyzed and, if any transformation was applied, a new *Pull Request* (PR) was 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 committed code changes.

We analyzed 140 free and open-source Android apps collected from *F-droid*<sup>11</sup>. Apps were selected by publish date (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 selected only one git service for the sake of simplicity. Apps in the dataset are spread in 17 different categories, as depicted in Figure 4.

Table 2 presents descriptive statistics for the source code and repository of the mobile applications in the dataset: number of lines of code (LOC), McCabe's Cyclomatic Complexity (CC), mean Weighted Methods per Class<sup>12</sup> (WMC), Lack of Cohesion of Methods<sup>13</sup> (LCOM) (Etzkorn et al.,

<sup>10</sup>F-droid repository is available at <https://f-droid.org> visited in August 17, 2019.

<sup>11</sup>Data was collected on Nov 27, 2016, and it is available here: <https://doi.org/10.6084/m9.figshare.7637402>

<sup>12</sup>Weighted Methods per Class (WMC) is the sum of the complexity of methods in a class.

<sup>13</sup>Lack of Cohesion of Methods (LCOM) is a software code metric that measures the correlation between class members and methods. Values fall between 0, indicating perfect cohesion, and 1, indicating a complete lack of cohesion.

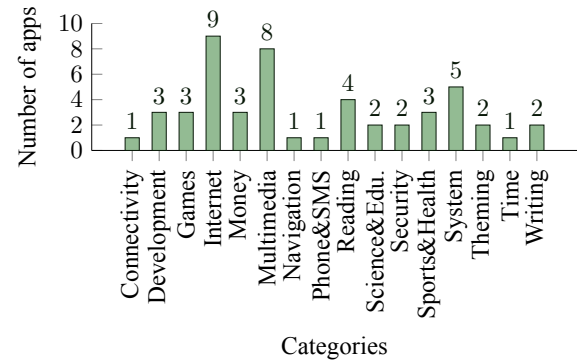


Figure 4. Number of apps per category in the dataset.

1998), number of Java files, number of XML files, number of Github Forks, Github Stars, and contributors. These metrics were collected using the static analysis tool *Designite*<sup>14</sup> and the *GitHub API v3*<sup>15</sup>.

The dataset comprehends very diverse mobile applications. It goes from very simple apps, such as *Storage-USB*<sup>16</sup>, with 13 LOC and complexity CC of 2, to large apps, such as *Slide*<sup>17</sup> with almost 400k LOC and complexity CC of 14631, or *Osmand*<sup>18</sup>, with over 300k LOC and complexity CC of 77889. The largest project in terms of Java files is *TinyTravelTracker* (1878), while *NewsBlue* is the largest in terms of XML files (2109). Most apps in the dataset have reasonable cohesion, with LCOM below 0.34 for 75% of the apps; apps with low/moderate cohesion were also analyzed, having LCOM values up to 0.67.

In total, we analyzed 2.8M lines of Java code (LOC) in 6.79GB of Android projects in 4.5 hours — 15103 XML files, and 15308 Java files.

## 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.

An example of the PRs submitted to the projects is illustrated in Figure 5. *Leafactor* performed the refactoring *ViewHolder* in the app *Slide*<sup>19</sup>, and developers successfully merged our PR. The full thread can be found in the *GitHub* project *ccrama/Slide* with reference #2346<sup>20</sup>.

<sup>14</sup>Designite's website: <http://www.designite-tools.com> visited in August 17, 2019.

<sup>15</sup>GitHub API v3's website: <https://developer.github.com/v3/> visited in August 17, 2019.

<sup>16</sup>*Storage-USB* basically launches Storage Settings directly from the apps drawer. Github repository: <https://github.com/enricocid/Storage-USB> visited in August 17, 2019.

<sup>17</sup>Slide is a browser for the social news forum Reddit. Github Repository: <https://github.com/ccrama/Slide> visited in August 17, 2019.

<sup>18</sup>*Osmand* is a navigation app. Github repository: <https://github.com/osmandapp/Osmand> visited in August 17, 2019.

<sup>19</sup>*Slide*'s website: <http://trikita.co/slide/> visited in August 17, 2019.

<sup>20</sup>PR of the *ViewHolder* of app *Slide*: <https://github.com/ccrama/>

**Table 2.** Descriptive statistics of projects in the dataset.

	LOC	CC	WMC	LCOM	Java Files	XML Files	Github Forks	Github Stars	Contributors
Mean	20350	3532	17.41	0.29	103	102	65	179	15
Min	13	2	1.00	0.00	0	4	0	0	1
25%	1444	271	11.14	0.23	13	23	3.75	7.75	2
Median	4641	946	15.20	0.27	38	48	9	24	3
75%	14795	3007	21.50	0.34	106	97	39	111	10
Max	388853	77889	82.82	0.67	1678	2109	1483	4488	323
Total	2869394	—	—	—	15308	15103	9547	26484	2162

**Table 3.** Summary of refactoring results

Refactoring	ViewHolder	DrawAllocation	Wakelock	Recycle	OLP*	Total
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`

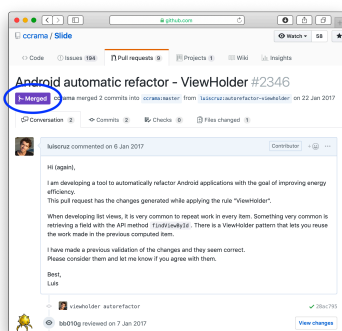
**Figure 5.** An example of pull request submitted to the Android project *Slide*.

Table 3 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. In addition, the table presents the combined results for the occurrence of any type of refactoring (*Total*).

*ObsoleteLayoutParam* was the most frequent refactoring. 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 refactoring. *Recycle* comes next, occurring in 23 projects (16%) with 58 refactorings. *DrawAllocation* and *Wakelock* only showed marginal impact.

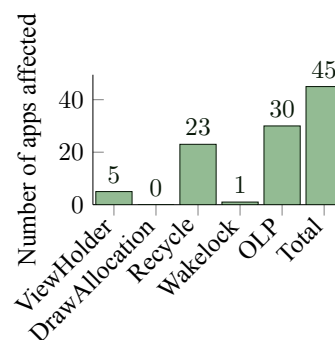
In addition, Figure 6 presents a plot bar summarizing the number of projects affected amongst all the studied refactorings.

The mobile application with a bigger incidence of refactorings was the Android application for the cloud platform *NextCloud*<sup>21</sup>. *Leafactor* has refactored two occurrences of *Recycle*, two of *ViewHolder*, and 6 of *ObsoleteLayoutParam*. In terms of the total number of refactorings, *QR Scanner*<sup>22</sup> was the app with a higher number of occurrences, with 35 occurrences of *ObsoleteLayoutParam*.

<sup>21</sup>*Slide/pull/2346* visited in August 17, 2019.

<sup>21</sup>*NextCloud*'s website: <https://nextcloud.com> visited in August 17, 2019.

<sup>22</sup>*QR Scanner*'s entry on Google Play: <https://play.google.com/store/apps/details?id=com.secuso.privacyfriendlyCodeScanner> visited in August 17, 2019.

**Figure 6.** Number of apps affected per refactoring.

For reproducibility and clarity of results, all the data collected in this study is publicly available<sup>23</sup>. In addition, all the PRs are public and can be accessed through the official repositories of the apps.

## 6 Discussion

Results show that an automatic refactoring tool can help developers ship more energy efficient apps. A considerable part of the 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. Reasons varied between seeing our PR as a critique of the programming skills of developers or simply because developers did not want to make changes in components of the app that were affected by the refactoring. Nevertheless, most developers were curious about the refactorings, and they recognized being unaware of their impact on energy efficiency. This is consistent with previous work (Pang et al., 2015; Sahin et al., 2014).

<sup>23</sup>Spreadsheet with all experimental results: <https://doi.org/10.6084/m9.figshare.7637402>.

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

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<sup>24</sup>. 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.

*Leafactor*, akin to *AutoRefactor*, applies the refactorings without prompting developers for confirmation. This is a common approach for simple refactorings. Nevertheless, in the case of energy code smells, a single refactoring entails changing several lines of code which the developer may not be able to interpret. During our experiments, this issue is mitigated since we submit a PR with a brief explanation of the code smell and the applied refactoring. It would be interesting to consider alternative approaches in which developers are informed or prompted while having their code refactored.

The code smell related to *ObsoleteLayoutParam* was found in a considerable fraction of projects (21%). This relates to the fact that app views are often created in an iterative process with several rounds of trial and error. Since some parameters have no effect under specific contexts, useless UI specification statements can go unnoticed by developers.

*Recycle* is frequent, too, being observed in 16% of projects. This smell 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<sup>25</sup>, it requires version 19 or earlier of Android SDK (introduced with Android 4.4 Kitkat). However, developers resort to a more verbose approach for backward 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 needs 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 refactoring due to its performance impact, and 2) many projects do not implement dynamic list views. *ViewHolder* is the most complex refactoring 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* is not intuitive.

Gainings on energy efficiency may vary depending on the application and the use cases in which they occur. Measuring the effective impact on energy consumption is not trivial as it requires a complicated setup. Previous work has found these refactorings to improve energy efficiency by up to 5% in real use case scenarios (Cruz and Abreu, 2017). Nonetheless, these refactorings are recommended by the official Android documentation<sup>26</sup> as best practices for performance.

A visible side effect of the refactorings featured by *Leafactor* is the questionable maintainability of the code introduced. Although the refactorings are implemented based on the official Android documentation, the resulting code is considerably longer and less intuitive for refactorings such as *ViewHolder* and *Recycle*. This is a threat to the adoption of energy-efficient practices in Android applications. Mobile frameworks should feature coding mechanisms aiming to improve energy efficiency without hindering code maintainability.

## 7 Related Work

The energy efficiency of mobile apps is being addressed with many different approaches. Some works opt by simplifying the process of measuring the energy consumption of mobile apps (Zhang et al., 2010; Pathak et al., 2012, 2011; Hao et al., 2013; Di Nucci et al., 2017; Couto et al., 2014). 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 (Li et al., 2014; Li and Halfond, 2014, 2015; Linares-Vásquez et al., 2017; Malavolta et al., 2017; Pereira et al., 2017).

Automatic detection of code smells for Android has been studied before. Fixing code smells in Android has shown gains up to 5% in energy efficiency (Cruz and Abreu, 2017). The code was manually refactored in six real apps and energy consumption was measured using a hardware-based power monitor. Our work extends this research by providing automatic refactoring to the resulting energy code smells.

The frequency of code smells in Android apps was studied in previous work (Hecht et al., 2015). Code smells were automatically detected in 15 apps using the tool *Paprika* 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 code smells and by applying automatic refactoring to resolve potential issues.

Previous work has also studied the importance of providing a catalog of bad smells that negatively influence the quality of Android applications (Reimann et al., 2014; Reimann and Aβmann, 2013). Although the authors motivate the importance of using automatic refactoring, their approach lacks an extensive implementation of their catalog. Related work has implemented 15 code-smells from this catalog proposed

<sup>24</sup>Example of a PR of refactorings on test code: <https://github.com/hidroh/materialistic/pull/828> visited in August 17, 2019.

<sup>25</sup>Documentation about the Java *try-with-resources* statement: <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html> visited in August 17, 2019.

<sup>26</sup>*ViewHolder* is documented here: <https://developer.android.com/training/improving-layouts/smooth-scrolling> visited in August 17, 2019.

by Reimann and A $\beta$ mann (2013) in an automatic refactoring tool, *aDoctor* (Palomba et al., 2017). In our work, we use this approach to improve the energy efficiency of Android applications.

Another work has focused exclusively on design patterns to improve the energy efficiency of iOS and Android mobile applications (Cruz and Abreu, 2019). However, no efforts were made regarding the automatic refactoring of the cataloged energy patterns. In our work, we implement automatic refactoring for five energy patterns. In addition, 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* (Gottschalk et al., 2012; Ebert et al., 2008). Eight different code smell detectors were implemented and validated with a navigation app. Fixing the code with automatic refactoring was discussed but not implemented. Besides, although studied code smells are likely to have an impact on energy consumption, no evidence was presented.

Previous work has used the event flow graph of the app to optimize resource usage (e.g., GPS, Bluetooth) (Banerjee and Roychoudhury, 2016). 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 (Sahin et al., 2014, 2016). However, these refactorings were not mobile-specific.

Besides refactoring source code, other works have focused on studying the impact of UI design decisions on energy consumption (Linares-Vásquez et al., 2017). 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 (Agolli et al., 2017). In our work, we strictly focus on changes that do not change the appearance of the app.

## 8 Conclusion

Our work presents the automatic refactoring tool *Leafactor* to improve the energy efficiency of Android application codebases. In an empirical study with 140 FOSS Android apps, we show the potential of using automatic refactoring tools to improve the energy efficiency of mobile applications. We have fixed 222 energy-related energy-related smells, improving the energy footprint of 45 Android applications. Results show that automatic refactoring can benefit developers to improve the energy efficiency for a considerable number of FOSS Android applications.

As future work, we plan to study and support more energy efficiency refactorings. In particular, some of the energy patterns studied in previous work (Cruz and Abreu, 2019; Reimann et al., 2014; Reimann and A $\beta$ mann, 2013) could help increase the usefulness of *Leafactor*. Besides, it would be interesting to explore the detection of energy-

related smells using dynamic analysis. Moreover, 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.

## Acknowledgements

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

- Agolli, T., Pollock, L., and Clause, J. (2017). Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 30–34. IEEE Press.
- Banerjee, A. and Roychoudhury, A. (2016). Automated refactoring of Android apps to enhance energy-efficiency. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, pages 139–150. ACM.
- Couto, M., Carção, T., Cunha, J., Fernandes, J. P., and Saraiva, J. (2014). Detecting anomalous energy consumption in Android applications. In *Brazilian Symposium on Programming Languages*, pages 77–91. Springer.
- Cruz, L. and Abreu, R. (2017). Performance-based guidelines for energy efficient mobile applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 46–57. IEEE Press.
- Cruz, L. and Abreu, R. (2018). Using automatic refactoring to improve energy efficiency of android apps. In *CIBSE XXI Ibero-American Conference on Software Engineering*.
- Cruz, L. and Abreu, R. (2019). Catalog of energy patterns for mobile applications. *Empirical Software Engineering*.
- Cruz, L., Abreu, R., and Rouvignac, J.-N. (2017). Leafactor: Improving energy efficiency of Android apps via automatic refactoring. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 205–206. IEEE Press.
- Di Nucci, D., Palomba, F., Protta, A., Panichella, A., Zaidman, A., and De Lucia, A. (2017). Petra: a software-based tool for estimating the energy profile of Android applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 3–6. IEEE Press.
- Ebert, J., Riediger, V., and Winter, A. (2008). Graph technology in reverse engineering—the tgraph approach. In *Proc.*



- 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*. Citeseer.
- Etzkorn, L., Davis, C., and Li, W. (1998). A practical look at the lack of cohesion in methods metric. In *Journal of Object-Oriented Programming*. Citeseer.
- Gottschalk, M., Josefiok, M., Jelschen, J., and Winter, A. (2012). Removing energy code smells with reengineering services. *GI-Jahrestagung*, 208:441–455.
- Hao, S., Li, D., Halfond, W. G., and Govindan, R. (2013). Estimating mobile application energy consumption using program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 92–101. IEEE.
- Hecht, G., Rouvoy, R., Moha, N., and Duchien, L. (2015). Detecting antipatterns in Android apps. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149. IEEE Press.
- Li, D. and Halfond, W. G. (2014). An investigation into energy-saving programming practices for Android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 46–53. ACM.
- Li, D. and Halfond, W. G. (2015). Optimizing energy of http requests in Android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 25–28. ACM.
- Li, D., Hao, S., Gui, J., and Halfond, W. G. (2014). An empirical study of the energy consumption of Android applications. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 121–130. IEEE.
- Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2014). Mining energy-greedy api usage patterns in Android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM.
- Linares-Vásquez, M., Bernal-Cárdenas, C., Bavota, G., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2017). Gemma: multi-objective optimization of energy consumption of guis in Android apps. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 11–14. IEEE Press.
- Malavolta, I., Procaccianti, G., Noorland, P., and Vukmirović, P. (2017). 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*, pages 35–45. IEEE Press.
- Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., and De Lucia, A. (2017). Lightweight detection of android-specific code smells: The adocor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 487–491. IEEE.
- Pang, C., Hindle, A., Adams, B., and Hassan, A. E. (2015). What do programmers know about the energy consumption of software? *PeerJ PrePrints*, 3:e886v1.
- Pathak, A., Hu, Y. C., and Zhang, M. (2012). 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*, pages 29–42. ACM.
- Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y.-M. (2011). Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168. ACM.
- Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P., and Saraiva, J. (2017). Helping programmers improve the energy efficiency of source code. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 238–240. IEEE Press.
- Reimann, J. and Aßmann, U. (2013). 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*, pages 321–326. IEEE Computer Society.
- Reimann, J., Brylski, M., and Aßmann, U. (2014). A tool-supported quality smell catalogue for Android developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung—MMSM*, volume 2014.
- Sahin, C., Pollock, L., and Clause, J. (2014). How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 36. ACM.
- Sahin, C., Pollock, L., and Clause, J. (2016). From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. *Journal of Systems and Software*, 117:307–316.
- Wilke, C., Richly, S., Gotz, S., Piechnick, C., and Aßmann, U. (2013). 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*, pages 134–141. IEEE.
- Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. (2010). 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*, pages 105–114. ACM.