Energy Refactorings for Android in the Large and in the Wild

Marco Couto
HASLab/INESC TEC
Universidade do Minho, Portugal
marco.l.couto@inesctec.pt

João Saraiva
HASLab/INESC TEC
Universidade do Minho, Portugal
saraiva@di.uminho.pt

João Paulo Fernandes
CISUC
Universidade de Coimbra, Portugal
jpf@dei.uc.pt

***Abstract*—Improving the energy efficiency of mobile applications is a timely goal, as it can contribute to increase a device's usage time, which most often is powered by batteries. Recent studies have provided empirical evidence that refactoring energy-greedy code patterns can in fact reduce the energy consumed by an application. These studies, however, tested the impact of refactoring patterns individually, often locally (e.g., by measuring method-level gains) and using a small set of applications.**

We studied the application-level impact of refactorings, comparing individual refactorings, among themselves and against the combinations on which they appear. We use scenarios that simulate realistic application usage on a large-scale repository of Android applications. To fully automate the detection and refactoring procedure, as well as the execution of test cases, we developed a publicly available tool called *Chimera*.

Our findings include statistical evidence that *i)* individual refactorings produce consistent gains, but with different impacts, *ii)* combining as much refactorings as possible most often, but not always, increases energy savings when compared to individual refactorings, and *iii)* a few combinations are harmful to energy savings, as they can actually produce more losses than gains.

We prepared a set of guidelines for developers to follow, aiding them on deciding how to refactor and consistently reduce energy.

***Index Terms*—Android, Energy, Code Patterns, Refactorings**

I. Introduction

Mobile devices such as smartphones or tablets are pervasive in our personal and professional activities, which is actually drawing significant attention for them to be energy efficient. Indeed, consumer satisfaction regarding a smartphone is highly dependent on its battery uptime [1]. For developers, battery uptime is also crucial, as abnormal drainage frequently justifies bad reviews in app stores [2]. Finally, the amounts of energy spent by mobile devices are heavily affecting sustainability [3].

In the context of Android - the largest mobile ecosystem - several works have focused on documenting energy-greedy programming patterns and on finding better alternatives for them. In fact, identifying and refactoring such code patterns to improve energy consumption has already presented promising research results [4]–[13]. These results, however, have essentially been validated by testing code patterns individually and often in a small set of applications (sometimes only in one).

In this paper, we consider 11 energy-greedy code patterns obtained from the literature, described in detail in Section III. We conduct a study over a large-scale repository of 600+ Android applications to understand the frequency of occurrence of such patterns. Within the 200+ applications where the patterns were detected, we studied the impact that replacing them, individually and combined, by their documented alternatives has on the energy consumption. Moreover, as we consider all the possible combinations of the individual patterns, this resulted in 400+ refactored applications under analysis.

To perform our study, we developed an extensible, fully automated framework called *Chimera*, which is able to detect and refactor the patterns. Each pattern is considered individually and is also combined with all the other patterns. *Chimera* also measures the energy consumed by an application in different simulated usage scenarios, before and after refactoring.

In summary, the main contributions of this work are:
1) An analysis of how energy-greedy patterns proposed in the literature are distributed over a large-scale repository of Android applications. This is described in Sections IV and V;
2) A reusable prototype of a pattern-oriented testing framework (*Chimera*), described in Section VI-B, for the detection, filtering, and refactoring of patterns in Android applications; it can also run a set of usage scenarios on such applications, while collecting metrics such as energy consumption;
3) An empirical study, described in Section VI, to assess the energy impact of applying refactorings. We analyze, for each code pattern and combination of patterns, the test results for the Android applications on which they occur, and compare the obtained gains between each pattern/combination.

Using the results of the empirical study referred in **3)**, we aim at answering the following research questions:

- **RQ1**: *Do refactorings consistently lead to energy savings?*
- **RQ2**: *Do all individual refactorings lead to energy savings of the same magnitude?*
- **RQ3**: *What are the refactorings that, individually or when combined, produce the higher energy savings?*
- **RQ4**: *When refactoring for energy efficiency, what approach should developers follow?*

In line with the literature, our findings confirm (in the large and in the wild) that refactorings consistently lead to energy improvements. Complementary, we have found statistical evidence that combining refactorings can produce higher energy savings, although in a few cases we discovered the opposite. As a final contribution of our work, we present a set of guidelines that developers can follow when aiming at refactoring

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

their applications to consistently improve their energy efficiency. All the study subjects, artifacts and results used and/or produced in our work are available in an online appendix[1].

The remaining sections of the paper are organized as follows: Section II describes related work; Section V-B discusses the technique used for automatically refactoring energy-greedy patterns; Section VII discusses threats to the validity of our study; Section VIII presents our conclusions and future work.

II. Related Work

Profiling, analyzing and improving the energy efficiency of software has become a very broad and prolific research area. With the increasing interest of developers in the subject [14], the extent of research works covers a wide variety of subjects, such as the energy impact of choosing different data structures [15]–[18], or languages [19], [20], how to model and even predict the energy consumption [21]–[23] or even point out energy leaks in the source code [24]. The energy impact of memoization [25], design patterns [26], code refactoring [27], [28], and even the testing phase [29], were also explored.

In the Android ecosystem, energy analysis has been also widely explored. In this context, several energy profilers have been proposed which operate at different program levels: *PowerTutor* [30] and *eProf* [31] at the application level, while *eCalc* [32] and *vLens* [33] at the byte code instruction and source line levels, respectively. To monitor energy consumption, however, these either require a device-dependent energy consumption model or an external measurement tool. Hence, in our work we used the Trepn tool, which can accurately be used on several Qualcomm-based Android devices [34] to profile energy consumption at the application level. This choice serves our intention of creating a framework that can work with as many devices as possible. Moreover, Trepn has been used in other energy-related research works [35]–[37].

Energy-related research in Android extends beyond the scope of profilers. Some works focused on classifying Android applications as being more/less energy efficient [35], by detecting well-known energy greedy APIs in the Android framework [12] using static analysis; others presented techniques to estimate energy consumption of code fragments such as methods [37], [38], and used that information to identify potential energy issues. Nevertheless, understanding how different programming strategies influence the energy consumption in Android is most likely the most explored subject in this area in the past decade [4]–[11], [39]–[41].

In [11] and [9], the authors focused on using static analysis to detect code patterns related to an energy inefficient use of resources, such as display or camera. Li et al. [10] proved that high memory usage, avoidable methods calls, and bundling of HTTP requests can potentially affect energy consumption.

Cruz et al. [8] studied how individually refactoring 5 performance-greedy code patterns reflects on the energy consumption of 6 real applications, concluding that in fact the refactorings produced energy savings; a tool was later developed to automatically refactor the patterns [6]. Carette et al. [7] did a similar study with 3 other patterns, which were analyzed individually and all together, in 5 different applications; the refactoring process is also automatic, and the authors reached similar conclusions: in the context on which the patterns were tested, the refactoring reduced the energy consumption.

The experiments performed in these studies were conducted over a small set of applications, hence the conclusions cannot be entirely generalized. Taking this into consideration, two works performed large-scale studies using different code patterns [4], [5]. In [4], the authors analyzed the energy impact of refactoring 9 patterns individually. Using a previously developed tool [42] to detect the patterns in 60 Android applications, they compared the energy consumed by methods containing each pattern before and after performing a manual refactoring. Identically, 5 object-oriented and 3 Android specific patterns were studied in [5], individually and over 20 applications, but with the goal of combining code quality with energy consumption when proposing refactorings.

All of the aforementioned works studied several patterns, yet no study has considered all patterns at once. Also, the patterns were analyzed under different conditions, and using different measurement tools/devices and whenever a large-scale study was conducted, the refactorings were performed manually. Finally, most works analyze the energy impact of the considered patterns individually, even though such patterns can occur simultaneously in the applications used for testing.

It is our strong belief that, with our work, we addressed all these issues. All code patterns for which the energy impact was already studied in Android were considered, and we provide an approach for automatic detecting and refactoring them. Our results include the energy impact of refactorings in several applications, both when applied alone and when combined with others. Such results were obtained from a large-scale study, where every test was conducted under the same conditions, using the same energy profiling tool.

A few of the patterns proposed in the past were excluded from our study. This was due to the fact that refactoring such patterns could not be done automatically, since it depends on their application context. For instance, in [5] the pattern *Long Parameter List* describes methods that have a long list of parameters, and the refactoring strategy consists of replacing the parameters with an object which includes all of them as instance variables; this implies the refactoring strategy should transform the method, but also look for every scenario and context on which the method is used and transform it accordingly, which, to be done automatically, is infeasible.

III. EGAPs - Energy Greedy Android Patterns

We have searched the literature for code patterns which have been tested for Android and that have proven to be energy inefficient; by an energy-inefficient pattern we mean a pattern for which an alternative exists, one that preserves the application's functionality, while consuming less energy. We have found 9

[1]Online appendix URL: http://marcocouto.gitlab.io/android-egaps

independent works that identify 11 energy-inefficient patterns and that propose alternatives for them [4]–[11], [41].

In the context of our work, and in its description in this paper, we shall refer to such a pattern as "**E**nergy **G**reedy **A**ndroid **P**attern", or **EGAP**.

In this section, we include a brief description of every EGAP we found, where for each one we indicate which research work(s) detected the pattern, while explaining what makes it energy greedy, what is the suggested alternative for it, and why such alternative consumes less energy.

A. *EGAP #1 - `Draw Allocation`*

This is the first of five EGAPs whose energy impact analysis was included in [6], [8]. The authors aimed at understanding how fixing code patterns detected by Android *lint*[2] can improve energy efficiency. *Lint*'s issues are divided into categories, such as Performance or Security. `Draw Allocation`, as well as the EGAPs described in Sections III-B, III-C, III-D, and III-E, is a **Performance** issue.

`Draw Allocation` occurs when new objects are allocated along with draw operations, which are very sensitive to performance. In other words, it is a bad practice to create objects inside the `onDraw` method of a class which extends a `View` Android component, as we see in the following snippet:

```
public class CloudMoonView extends View {
  @Override
  protected void onDraw(Canvas canvas) {
    RectF rectF1 = new RectF(); ✘
    ...
    if(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
      ...
} }
```

The recommended alternative for this EGAP, as of [6], [8], is to move the allocation of independent objects outside the method, turning it into a static variable, as shown next:

```
public class CloudMoonView extends View {
  RectF rectF1 = new RectF(); ✔
  @Override
  protected void onDraw(Canvas canvas) {
    ...
    if(!clockwise) {
      rectF1.set(X2-r, Y2-r, X2+r, Y2+r);
      ...
} }
```

B. *EGAP #2 - `Wakelock`*

`Wakelock` is the second Android *lint* performance issue [4], [6], [8], [11]. Basically, *lint* detects whenever a wake lock, a mechanism to control the power state of the device and prevent the screen from turning off, is not properly released, or is used when it is not necessary.

The following snippet shows an example of a wake lock being acquired, but not released when the activity pauses.

```
public class DMFSetTempo extends Fragment {
  PowerManager.WakeLock wakeLock;

  public void onClickBtStart(View view) {
    wakelock.acquire();  ✔
  }

  @Override()
  public void onPause() { super.onPause(); ✘ }
  }
}
```

[2]*Lint* is a code analysis tool, provided by the Android SDK, which reports upon finding issues related with the code structural quality. Website: developer.android.com/studio/write/lint

The alternative here would be to simply add a `release` instruction as shown next:

```
public class DMFSetTempo extends Fragment {
  PowerManager.WakeLock wakeLock;

  public void onClickBtStart(View view) {
    wakelock.acquire();    ✔
  }

  @Override()
  public void onPause() {
    super.onPause();
    if (wakeLock.isHeld()) wakeLock.release();  ✔
  }
}
```

C. *EGAP #3 - `Recycle`*

`Recycle` is another Android *lint* performance issue [6], [8]. It detects when some collections or database related objects, such as `TypedArrays` or `Cursors`, are not recycled nor closed after being used. When this happens, other objects of the same type cannot efficiently use the same resources.

The following snippet shows a `Cursor` instance being used without being recycled:

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db = this.getReadableDatabase();

  Cursor cursor = db.query(TABLE_FAV, new String[] { ... };
  ...
  return summoner;  ✘
}
```

The alternative in this case would be to include a `close` method call before the method's return:

```
public Summoner getSummoner(int id) {
  SQLiteDatabase db = this.getReadableDatabase();

  Cursor cursor = db.query(TABLE_FAV, new String[] { ... };
  ...
  c.close();   ✔
  return summoner;
}
```

D. *EGAP #4 - `Obsolete Layout Parameter`*

The fourth Android *lint* performance issue [6], [8], `Obsolete Layout Parameter`, is the only one that is not Java-related. The view layouts in Android are specified using `XML`, and they tend to suffer several updates. As a consequence, some parameters that have no effect in the view may still remain in the code, which causes excessive processing at runtime. The alternative is to parse the `XML` syntax tree and remove these useless parameters.

The next snippet shows an example of a view component with parameters that can be removed:

```
<TextView android:id="@+id/centertext"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="remote files"
  layout_centerVertical="true"    ✘
  layout_alignParentRight="true"> ✘
</TextView>
```

E. *EGAP #5 - `View Holder`*

`View Holder` is the last Android *lint* performance issue [6], [8], whose alternative intends to make a smoother scroll in *List Views*. The process of drawing all items in a *List View* is costly, since they need to be drawn separately. However, it is possible to make this more efficient by reusing data from already drawn items, which reduces the number of calls to `findViewById()`, known to be energy greedy [12].

In order to better describe this EGAP, we introduce the following snippet:

```
public View getView(int pos, View cView, ViewGroup par) {
  LayoutInflater inflater = (LayoutInflater) context
    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);

  cView = inflater.inflate(R.layout.apps, par, false);
  TextView txt=(TextView) cView.findViewById(R.id.label); ❶
  ImageView img=(ImageView) cView.findViewById(R.id.logo);❷
  return row;
}
```

Every time `getView()` is called, the system searches on all the view components for both the `TextView` with the id "label" (❶) and the `ImageView` with the id "logo" (❷), using the energy greedy method `findViewById()`. The alternative version is to cache the desired view components, with the following approach:

```
static class ViewHolderItem {
  TextView txtView; ImageView imgView;
}

public View getView(int pos, View cView, ViewGroup par) {
  ViewHolderItem hld; LayoutInflater inflater = ...

  if (cView == null) {                                    ❸
    cView = inflater.inflate(...);
    hld = new ViewHolderItem();
    hld.txtView = (TextView) cView.findViewById(...);     ❹
    hld.imgView = (ImageView) cView.findViewById(...);    ❺
    cView.setTag(hld);
  } else {
    hld = (ViewHolderItem) cView.getTag();                ❻
  }
  TextView txt = hld.txtView;  ImageView img = hld.imgView;
  ...
}
```

Condition ❸ evaluates to true only once, which means instructions ❹ and ❺ execute once, i.e., `findViewById()` executes twice, and its results are stored in the `ViewHolderItem` instance. The following calls to `getView()` will use cached values for the view components `txt` and `img` (❻).

F. EGAP #6 - `HashMap Usage`

This EGAP is related to the usage of the `HashMap` collection [4], [5], [7], [41]. In fact, as stated in the Android documentation page, the usage of `HashMap` is discouraged, since the alternative `ArrayMap` is allegedly more energy-efficient, without decreasing the performance of `map` operations[3].

The alternative is to simply replace the type `HashMap`, whenever it is used, with `ArrayMap`.

G. EGAP #7 - `Excessive Method Calls`

Unnecessarily calling a method can penalize performance, since a call usually involves pushing arguments to the call stack, storing the return value in the appropriate processor's register, and cleaning the stack afterwards. This penalty was explored by [7], [10], showing that the energy consumption in Android applications can be decreased by removing method calls inside loops that can be extracted from them. An example of an extractable method call would be one which receives no arguments, and is accessed by an object that is not altered in any way inside the loop.

The alternative is to replace the method call by a variable that is declared outside the loop, and is initialized with the return value of the method call extracted.

[3]*ArrayMap* documentation: http://bit.ly/32hK0y9.

H. EGAP #8 - `Member Ignoring Method`

This EGAP addresses the issue of having a non-static method inside a class, and which could be static instead [4], [7], i.e., it does not access any class fields, it does not directly invoke non-static methods, and it is not an overriding method. Static methods are stored in a memory block separated from where objects are stored, and no matter how many class instances are created throughout the program's execution, only an instance of such method will be created and used. This mechanism helps in reducing energy consumption.

I. EGAPs #9, #10 & #11 - `Resource Leak`

`Resource Leak` [9], [11] simultaneously refers to three EGAPs which are all particular cases of `Wakelock`, where a system resource is not properly released. Here, we consider the *Sensor*, *Camera*, and *Media* resources, which differ from `Wakelock` in the way resources are released. Since improving EGAPs on different resources can produce different energy gains, we decided to separate EGAPs *#2* from *#9*, *#10* & *#11*.

IV. The Android Application Repository

In order to study the practical impact of replacing energy greedy patterns with their documented alternatives, a large-scale repository of Android applications from the real world was deemed necessary.

During our research, we came across large repositories of open-source Android applications such as the F-Droid catalogue[4]. However, using them in the context of our work would require additional effort in collecting the applications' source code: we would have to collect, for each application, the available information, parse it to find the source repository (which can be stored in different platforms, such as GitHub or GitLab), and download it. This additional step would be time consuming, and we wanted to avoid it.

Therefore, we followed an alternative approach to obtain a representative set of Android applications: we have taken the MUSE repository [43], [44] as a starting point for our work.

MUSE[5] is a repository of Java projects, collected from public repositories contained in well known source code platforms, such as GitHub, Bitbucket or Apache. In addition to the projects' source code collection, MUSE has an associated database where, for each project, information regarding its source code is also available. For example, it is possible to know how many files each project has, what import statements are used and what classes are declared in each file, what methods and variables are declared in each class, and so on.

Since Android applications are mainly Java projects, we were able to query MUSE and search for projects containing Android API components. The filtering strategy was to look at the import statements of each project. If there were any Android API imports, then it was and Android application project. We were able to obtain **609** buildable and executable Android projects, that we take as study subjects in this work.

[4]F-Droid webpage: f-droid.org.
[5]MUSE webpage: https://muse-portal.net/.

In order to better understand the dimension of the Android projects repository, we computed, for each one of the 609 Android projects, the number of lines of code, Java files, XML files, classes, and methods. The results of this analysis are summarized in Table I where we include, for each metric, the maximum, minimum and average values.

TABLE I
APPLICATIONS REPOSITORY METRICS

	Java Files	*XML Files*	*Classes*	*Methods*	*LOC*
Min	2	1	2	1	22
Max	3,492	4,929	4,267	39,511	668,085
Average	73	407	65	524	10,822
TOTAL	**44,959**	**248,385**	**39,926**	**319,636**	**6,590,636**

V. AUTOMATIC EGAP DETECTION AND TRANSFORMATION ON ANDROID SOURCE CODE

In this section, we describe the methodology used for detecting and transforming the patterns described in Section III, in the large-scale repository of Android applications described in Section IV. We start by explaining the techniques used to detect EGAPs in all the applications, and how we process the output from this analysis (Section V-A). Next, we present our approach for the automatic transformation of such patterns, along with a discussion regarding its applicability (Section V-B). Finally, we study how individual and combined EGAPs are distributed in the analyzed applications, and we present the final list of (individual and combined) EGAPs that we managed to analyze in this paper (Section V-C).

A. Detection Methodology

Detecting a pattern in the source code of an Android application can be achieved by: i) parsing it into an abstract syntax tree (AST), ii) traversing through such AST, and iii) reporting where the pattern is found. These steps have already been abstracted within the *lint* tool which allows the possibility of creating rules for the detection of such patterns[6].

In fact, *lint* already includes rules which allow the detection of EGAPs **#1 - #5**. So, we needed to create custom rules for the detection of the remaining 6 EGAPs. Furthermore, the *lint* tool enables the selection of which rules to apply, which allowed us to consider the detection of "our" 11 EGAPs only, avoiding further analysis. Finally, it also contains a built-in report system which automatically produces an XML file indicating where a pattern is found in the code.

Defining a custom *lint* rule consists of creating a class where we first define a set of properties, such as the rule description, category, or severity. Along with such properties, we must define the rule's workflow: what AST node types must be analyzed, what to do before/after traversing through the AST, and most importantly what visitor to use in each traversal.

A visitor goes through the AST in a bottom-up approach, running the respective *visit* method when it finds a node signaled as analyzable. All the EGAP-related properties are inferred in *visit* methods; when an EGAP is found we use the built-in report system to store its location. The next snippet shows how we implemented one such method, namely to detect the `HashMapUsage` EGAP, and that reports when a method variable is declared with the *HashMap* type:

```
private void checkAndReport(PsiVariable var) {
  if (var.getType() == null) return;
  String varType = var.getType().getCanonicalText();
  PsiExpression init = var.getInitializer();

    if (varType.startsWith(mHashMapClass)) {
      Reporter.reportIssue(mContext, ISSUE, var);
    } else if (varType.startsWith(mMapClass)
      && init != null && init.getType() != null) {
        String initTp = init.getType().getCanonicalText();
        if (initTp.startsWith(mHashMapClass)) {
          Reporter.reportIssue(mContext, ISSUE, var);
        }
}  }
```

Having at hand the custom rules for detecting all EGAPs, we then ran the *lint* tool on each of the 609 considered applications and collected the report files. The *lint* report consists of a list of EGAP occurrences within an application, with information regarding where in a file was the EGAP found. For each application, we parsed the report file to find which EGAPs were detected (and the number of occurrences). We group and store this information in `JSON` format.

In order to validate the accuracy of our detection methodology, we used a set of sample applications where all the patterns were manually injected, in all contexts on which they can occur[7]. After *lint* analyzed them, we manually searched for both false positives and negatives. We confirmed that all patterns were properly detected, with no inconsistencies found.

B. Automatic Refactoring Approach

To automate the transformation of Android EGAPs, we reused the ***AutoRefactor***[8] system: a well-known Java refactoring framework. This is an open-source Eclipse IDE plugin and it supports refactoring of Android applications, which was already used to refactor a small-set of energy-greedy Android patterns [6], [45]. In *AutoRefactor* a refactor is concisely defined as a set of source code transformation rules. Cruz et al. [8] defined these rules to express refactorings for EGAPs #1 - #5. We extend this work with transformation rules for the remaining EGAPs, and we integrate all refactorings in the *command line interface* (CLI) mechanism of Autorefactor.

Following the same approach as Cruz et al. [8], we designed the transformations to be generic without compromising the applications integrity. This means that although we detect all EGAPs reported in the literature that may occur in an Android application, it was not always possible to safely apply the corresponding refactoring automatically. This happens when the transformation does not preserve the semantics of the pplication (e.g., in EGAP #6 when the constructor of a *HashMap* receives a *Map* object, it is not possible to change it into an *ArrayMap*, as the former has no such constructor).

We were able to detect at least one EGAP in **239** applications, and overall we detected **71** different combinations of

[6]More information can be found here: http://bit.ly/2Mh7aPF.

[7]For instance, for EGAP *#6* we injected references to `HashMap` in variable types, method arguments, cast expressions, and instance creation statements.

[8]*AutoRefactor* webpage: autorefactor.org.

EGAPs. Out of these 71, we were able to refactor, at least in one application, **44** combinations, which then consist of our (EGAP combination) study base. Overall, we obtained **416** different applications where at least one refactoring was applied, which consist of our (application) study base[9].

This represents a considerable larger study base than reported in the literature.

C. Results and Discussion

Table II summarizes the results of the EGAP detection and refactoring phases. The first column contains the total set of combinations that were found in at least one application, and the second column shows the number of applications on which each combination was detected. Columns **Max per app.** and **Min per app.** respectively show the highest and lowest number of times that EGAPs in the combination were detected in an application. For example, for combination *[#1, #6]*, the application in which the highest number of (*#1+#6*) EGAPs were found totaled 5 occurrences (4 of EGAP *#1* and 1 of EGAP *#6*). Finally, column **# Apps Refactored (%)** shows the success rate in refactoring that combination[10].

TABLE II
RESULTS OF EGAP DETECTION AND REFACTOR PHASES

EGAP Combination	#Apps Detected	Max per app.	Min per app.	# Refactored apps (%)
[#1]	12	39	1	5 (41.7%)
[#2]	2	1	1	1 (50%)
[#3]	29	11	1	5 (17.2%)
[#4]	40	39	1	39 (97.5%)
[#5]	5	6	1	4 (80%)
[#6]	84	21	1	60 (71.4%)
[#7]	111	53	1	50 (45%)
[#8]	172	155	1	101 (58.7%)
[#1, #6]	5	[4, 1]	[1, 1]	2 (40%)
[#1, #7]	5	[1, 8]	[1, 1]	2 (40%)
[#1, #8]	7	[39, 5]	[2, 2]	1 (14.2%)
[#3, #4]	5	[5, 3]	[1, 1]	2 (40%)
[#3, #6]	17	[7, 21]	[1, 1]	2 (11.8%)
[#3, #8]	20	[7, 155]	[1, 1]	1 (5%)
[#4, #5]	2	[4, 1]	[1, 2]	2 (100%)
[#4, #6]	17	[1, 20]	[1, 1]	11 (64.7%)
[#4, #7]	20	[39, 4]	[1, 1]	10 (50%)
[#4, #8]	26	[2, 30]	[1, 1]	7 (26.9%)
[#5, #6]	3	[6, 4]	[2, 1]	2 (66.6%)
[#5, #7]	3	[1, 8]	[1, 1]	2 (66.6%)
[#5, #8]	3	[6, 8]	[2, 6]	1 (33.3%)
[#6, #7]	47	[21, 53]	[1, 1]	12 (25.5%)
[#6, #8]	62	[21, 155]	[1, 1]	23 (37.1%)
[#7, #8]	76	[53, 155]	[1, 1]	25 (32.8%)
[#1, #6, #7]	3	[3, 1, 5]	[1, 1, 1]	1 (33.3%)
[#1, #6, #8]	4	[3, 1, 18]	[1, 1, 3]	1 (25%)
[#1, #7, #8]	5	[3, 5, 18]	[1, 1, 6]	2 (40%)
[#3, #4, #6]	4	[1, 1, 20]	[5, 3, 2]	2 (50%)
[#4, #5, #6]	1	[4, 1, 6]	[4, 1, 6]	1 (100%)
[#4, #5, #7]	2	[4, 1, 8]	[1, 2, 4]	1 (50%)
[#4, #5, #8]	1	[4, 1, 10]	[4, 1, 10]	1 (100%)
[#4, #6, #7]	13	[1, 20, 17]	[1, 1, 1]	5 (38.5%)
[#4, #6, #8]	16	[1, 20, 18]	[1, 1, 2]	7 (43.8%)
[#4, #7, #8]	16	[2, 12, 30]	[1, 1, 2]	5 (31.2%)
[#5, #6, #7]	1	[1, 6, 8]	[1, 6, 8]	1 (100%)
[#5, #6, #8]	3	[6, 4, 8]	[2, 1, 6]	2 (66.6%)
[#5, #7, #8]	1	[1, 8, 10]	[1, 8, 10]	1 (100%)
[#6, #7, #8]	40	[21, 53, 155]	[1, 1, 1]	8 (20%)
[#1, #6, #7, #8]	3	[3, 1, 5, 18]	[1, 1, 1, 6]	1 (33.3%)
[#4, #5, #6, #7]	1	[4, 1, 6, 8]	[4, 1, 6, 8]	1 (100%)
[#4, #5, #6, #8]	1	[4, 1, 6, 10]	[4, 1, 6, 10]	1 (100%)
[#4, #5, #7, #8]	1	[4, 1, 8, 10]	[4, 1, 8, 10]	1 (100%)
[#4, #6, #7, #8]	12	[1, 20, 17, 18]	[1, 1, 1, 2]	3 (25%)
[#4, #5, #6, #7, #8]	1	[4, 1, 6, 8, 10]	[4, 1, 6, 8, 10]	1 (100%)

[9]Note that the same (original) application is being counted more than once, in case more than one combination of EGAPs was actually applied to it.

[10]Combinations with a success rate of 0% are available in the supplementary material and were omitted due to space limitations.

From the set of 609 analyzed applications, we observed that:
1. EGAPs #9, #10 and #11 were not found in any application. Despite their relevance described in the literature [9], [11], this suggests that the contexts where they were found might not be generally representative;
2. on **370** of them ($\approx 60\%$) none of the 11 EGAPs were detected. This suggests that there still might be room to propose new energy greedy source code patterns or that potentially a significant amount of application developers already follow the best practices for energy efficient code;[11]
3. EGAPs #6, #7, and #8 are the most frequent, either individually or combined. This might be related to the fact that these three particular EGAPs are not exclusively related with the Android API. Indeed, any Java application can use an *HashMap*, and this seems to confirm that not declaring a method as static when it is possible to do so, or unnecessarily calling a method inside a loop (e.g., to improve readability), is common. Nonetheless, it is important to notice that not only Google itself suggests to avoid using the *HashMap* collection for performance reasons, but it was also already proved that this collection can not only be energy-greedy in Android applications, but also in typical Java applications [17], [46].

VI. ANALYZING THE ENERGY IMPACT OF REFACTORING EGAPS

The goal of our study is to understand the energy impact of refactoring EGAPs. In this section, we explain the procedure we followed: the tools and methodology used, the data collected, and how it was used to reason about energy impact.

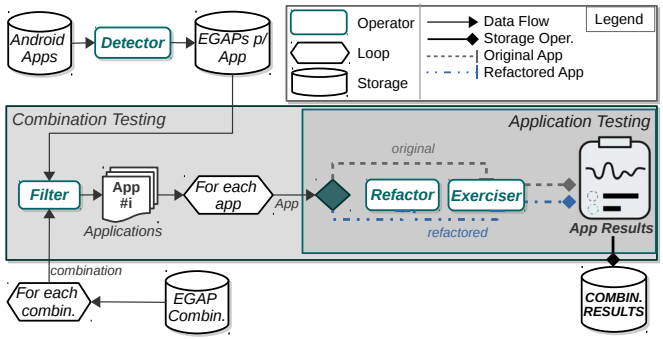

Fig. 1. Case Study Pipeline

A. Experimental Setup

Our experiments were performed in three factory-reseted Nexus 5 mobile devices running Android 6.0.1. Energy consumption data was obtained using the (same version of the) Qualcomm Trepn profiler[12], a profiling application that has been used in several energy-related studies [35]–[37].

We divided the workload through all devices in a way that all applications to be tested for each combination of EGAPs (both before and after refactoring) are executed in the same

[11]In the latter case, note that developers may already request EGAPs #1 - #5 to be automatically signaled upon the application build.

[12]Trepn webpage: developer.qualcomm.com/software/trepn-power-profiler.

device. This ensures not only that the comparison between the different versions of the same application, but also the analysis of the impact of one EGAP combination, are made using data gathered from the same device.

During each test execution, the only applications running were, aside from OS-related Android applications and services, the application under test and Trepn (running as a service, in background). Moreover, we kept all devices in airplane mode and with minimum screen brightness, to reduce as much as possible the measurement noise.

The EGAPs we considered can be found in different components in an Android application (e.g., the `Wakelock` is associated only with *Activities*, while the `HashMap Usage` can occur anywhere in the application). Thus, we decided to test the applications where the EGAPs were transformed by running different usage scenarios on them.

B. Experimental Pipeline

For the purpose of testing the energy effects of refactoring an EGAP, we developed a testing framework which incorporates EGAP detection and refactoring. This framework is also capable of automatically running a predefined set of usage scenarios, storing the results of each scenario, and grouping the results per application and EGAP. The framework implements the execution pipeline depicted in Figure 1.

The execution pipeline contains 4 operators responsible for performing independent tasks, and needs to receive as input both the complete set of EGAP combinations (included in Table II) and the list of EGAPs detected in each application. Next, we explain each pipeline operator.

The *Detector* operator detects EGAPs in an application, and stores its results so they can be further utilized. For each existing EGAP combination C_i, *Filter* uses the output from *Detector* to filter and gather the applications containing all EGAPs in C_i. It returns both C_i and the source code of the filtered applications.

Each application selected by *Filter* is forked into two versions: the *original*, and the *refactored*. The latter version goes through the *Refactor* operator, which is responsible for applying the refactorings for the EGAPs in C_i. This operator's outputs are the 2 application versions, along with the number of times each detected EGAP was refactored.

Using the output from *Refactor*, *Exerciser* first checks the list of refactored EGAPs. If there is at least one of the EGAPs in C_i (the combination under analysis) that is not contained in the list of refactored EGAPs, then the application will not be executed. If *Refactor* completes successfully, then the 2 application versions will be executed to produce the results.

To simulate the application's usage scenarios, we used the *Android MonkeyRunner*[13]. We chose this tool because it provides a very convenient method to inject events on an application, and by this to simulate usage scenarios without knowing the application context. In fact, a recent study shows that randomly injecting events over Android applications provides a way of testing them and assure the best ratio between coverage and effort needed for the setup [47]. Considering we are analyzing a substantial set of applications, it would be unpractical to build context-dependent usage scenarios for each. Therefore, we designed **25 usage scenarios** that can be applied to any application. Each scenario consists of 2 parameters that are given to *MonkeyRunner*: the application package, and a randomly generated sequence of **25** events to be triggered. Each sequence is unique, in order to simulate different execution paths of the application. The scenarios run on the original and the refactored version are exactly the same.

In each scenario execution, *Exerciser* opens the application, waits **5 seconds** (warm-up), and then starts the *MonkeyRunner* events and the measurement procedures. When *MonkeyRunner* finishes, *Exerciser* stops measuring, closes the app, stores the collected values, and waits another **5 seconds** (cool-down). This procedure is repeated both for the *original* and the *refactored* versions of an application.

C. Results and Discussion

The experiments described in the previous section took over **500 hours** to finish. Once finished, we obtained results for **416** applications. For each application, and for each of its 25 test scenarios, the experiment produced *i)* the number of times each EGAP was refactored and *ii)* the energy consumed by the *original* (E_O) and *refactored* (E_T) versions. Using this information, we were able to calculate the **gain** obtained in each test, which is the percentage difference between E_O and E_R, calculated using the formula: $\frac{(E_O - E_R)}{E_O} \times 100$. A negative value means a **loss** of energy in a test.

The results of the experiments are summarized in Figure 2. For each EGAP (or combination), we included a bar plot comparing the proportion of usage scenarios which resulted in gains with the ones that resulted in losses, with the respective number in front of each bar. The boxplots indicate how the exact gains/losses values are distributed[14]. Combining these two plots, we can observe not only if a certain combination had overall more gains/losses, but also if the impact was higher.

We can see that, when refactoring individual patterns, the number of tests that resulted in gains is always greater than the number of losses. Moreover, as the boxplots confirm, the gain values are always considerably higher than the loss values. In line with the literature, this confirms that refactoring such patterns individually consistently leads to energy savings.

Looking at combinations of EGAPs, the majority of them do clearly have higher gains than losses. Nevertheless, in a few cases the gains do not clearly surpass the losses, or are even lower. For instance, combination *[#4, #5]* has the exact same number of tests with gains and losses, and the boxplots are very similar, whereas *[#4, #8]* has more tests which resulted in gains, but the boxplots indicate a slight tendency to favor the losses. The more interesting scenario, however, occurs for combination *[#4, #5, #6, #7]*: the distribution of gain values demonstrates that they were considerably low (all below $\approx$3%), and are clearly surpassed by the losses.

[13] Android MonkeyRunner webpage: http://bit.ly/2VzPFAF.

[14] Outliers were excluded for the purpose of aiding data visualization.

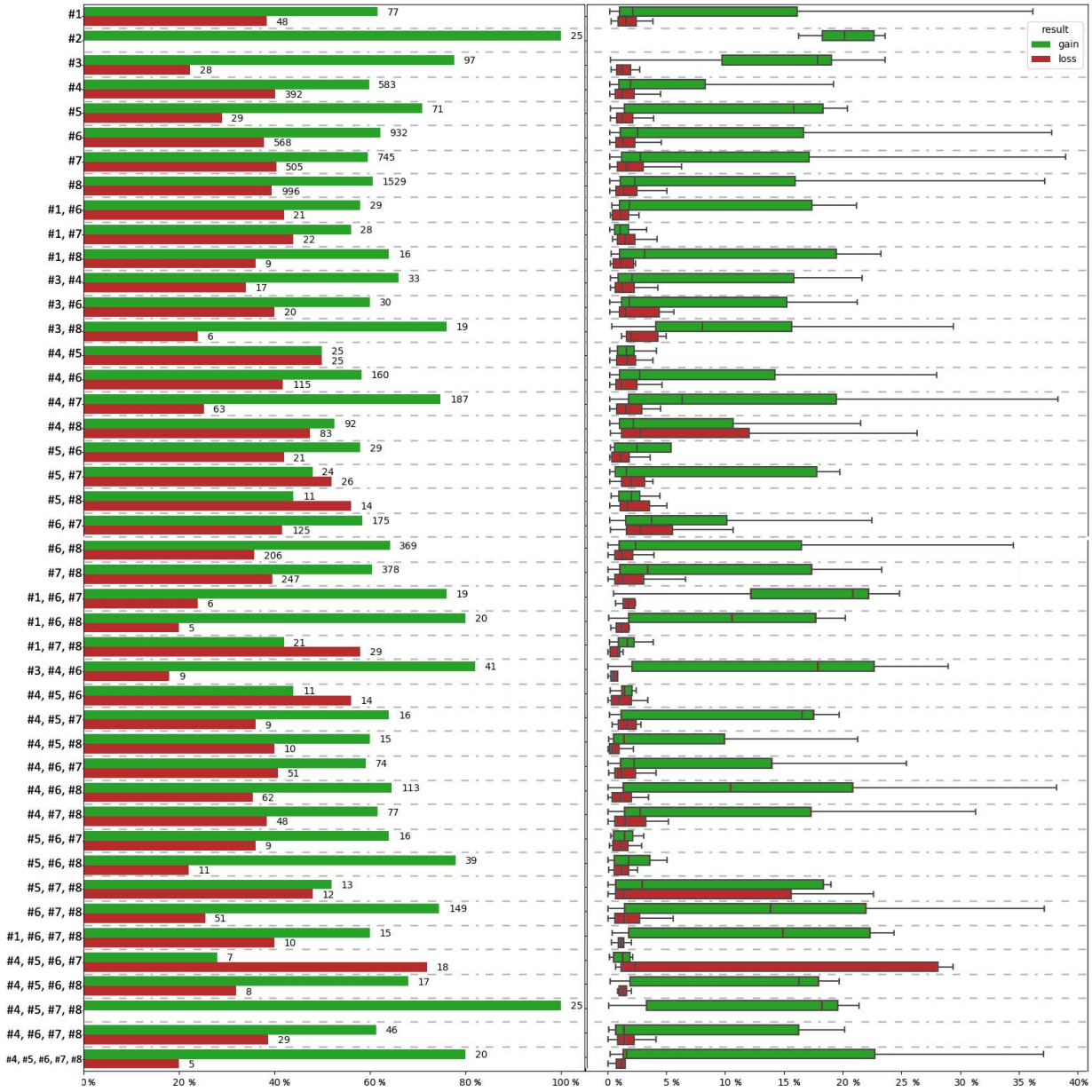

Fig. 2. Overview: Experimental Results

Based on our results, we can safely state that **when refactoring patterns individually, the gains are consistently higher than the losses, but the same cannot be concluded for a few combinations of patterns**, hence answering **RQ1**.

In order to answer **RQ2**, we examined, for each individual refactoring, how similar were its gains/losses when compared to the other 7. We ran a *Mann–Whitney U test*, which is used to assess if two independent samples (in our case, a set of gains/losses) are selected from populations having the same distribution. If H_0, the null hypothesis, is rejected, the gains/losses of two refactorings are significantly different. Also, the resulting test statistics can assess if a value from one sample is likely to be higher or lower than one randomly taken from the other, and by how much. We consider $\alpha = 0.05$, so we can reject the null hypothesis with 95% confidence.

TABLE III
RESULTS OF MANN–WHITNEY U TEST FOR INDIVIDUAL COMBINATIONS

	#1		**#2**		**#3**		**#4**		**#5**		**#6**		**#7**		**#8**	
	ρ	*stats*	*ρ*	*stats*	*ρ*	*stats*	*ρ*	*stats*	*ρ*	*stats*	*ρ*	*stats*	*ρ*	*stats*	*ρ*	*stats*
#1	–	–	0	-7.19	0	-5.64	**0.98**	0.03	0	-2.71	**0.41**	-0.83	**0.75**	-0.32	**0.69**	-0.41
#2	0	7.19	–	–	0	5.49	0	7.46	0	6.38	0	7.66	0	7.14	0	7.40
#3	0	5.64	0	-5.49	–	–	0	7.19	0	2.82	0	6.86	0	6.71	0	6.91
#4	**0.98**	-0.03	0	-7.46	0	-7.19	–	–	0	-3.32	**0.05**	-1.97	**0.39**	-0.86	**0.34**	-0.96
#5	0	2.71	0	-6.38	0	-2.82	0	3.32	–	–	0	2.76	0	2.92	0	2.96
#6	**0.41**	0.83	0	-7.66	0	-6.86	**0.05**	1.97	0	-2.76	–	–	**0.28**	1.09	**0.23**	1.20
#7	**0.75**	0.32	0	-7.14	0	-6.70	**0.39**	0.86	0	-2.92	**0.28**	-1.09	–	–	**0.98**	0.03
#8	**0.69**	0.41	0	-7.40	0	-6.91	**0.34**	0.96	0	-2.96	**0.23**	-1.20	**0.98**	-0.03	–	–

The test results are depicted in Table III. Each row represents an EGAP, and it contains the *p-value* (ρ) and *test statistics* ($stats$) resulting from testing the gains/losses of that EGAP against the ones of the EGAP in the corresponding

column (*p-values* below 0.01 were rounded to 0).

From the obtained *p-values*, most gains are expected to be quite different between refactorings (rejected H_0). In a few cases, a high *p-value* is observed; here, the expected gains are very similar for both refactorings. We see that, e.g., `DrawAllocation` (#1) is expected to produce similar gains as `ObsoleteLayoutParam` (#4) and `ExcessiveMethodCalls` (#7) (*p-values* of 0.98 and 0.75).

When H_0 is rejected, the $stats$ value gives us an indication on what to expect from the results of two refactorings. For instance, comparing `Recycle` (#3) and `DrawAllocation` (#1), we have a $stats$ value of 5.64, and when comparing with `Wakelock` (#2) the value is -5.49. This means that the average gain from refactoring `Recycle` is expected to be about 5.64 times bigger than the ones obtained from refactoring `DrawAllocation`, but at the same time about 5.49 times lower when compared with `Wakelock`.

In conclusion, to answer **RQ2**, **refactoring individual EGAPS most often, but not always, leads to energy savings with significantly different orders of magnitude**.

With **RQ3**, we first sought to understand whether it is always preferable to apply, in an application, all the refactorings that are possible. For this, we compared every single refactoring, with all the combinations where such refactoring is included. For fairness, we considered only the applications where both the individual refactoring and the combinations which include it are available.

Again, a *Mann–Whitney U test* was used to compare the gains of every individual refactoring R_i against all combinations C_i where it occurs. If H_0 is rejected, there is statistical evidence that the gains obtained from applying R_i are either consistently higher or lower than the ones obtained from applying C_i. If so, we also wanted to quantify how big that difference is. We calculated the *Cohen's d* value between gains obtained from R_i and C_i, and we used the Sawilowsky [48] suggested thresholds of 0.2, 0.5, 0.8, 1.2, and 2, which respectively translates to small, medium, large, very large and huge effect size.

The results we obtained are depicted in Figure 3[15]. Each individual refactoring on the left is connected to the combinations on the right on which the statistical test suggested to reject H_0 (i.e. where the gains between them are undeniably different, when applied on the same applications). A green connection reveals a tendency for the gains to be higher for the combination on the right; we shall refer to such scenarios as *positive effects*. Similarly, yellow, orange, or red connections indicate scenarios where the gains tend to be lower, and we shall call them *negative effects*. The color intensity and the connection width express the effect size.

If it would be preferable to apply as much refactorings as possible, then *all* connections should express *positive effects*, or at least the *negative* ones should not exist (which would ultimately mean that there was no strong evidence). In practice, out of the existing 38 connections, 22 are *positive* and 16 are *negative*, with the latter having an overall more relevant effect size. In conclusion, and to provide the answer to **RQ3**, we can say that **combinations that contain *positive* connections are the ones with higher gains**, as such connections reflect the existence of higher and consistent gains. Still, the existence of *negative* connections provides evidence that **the gains obtained by some individual refactorings can be significantly reduced when combined with others**.

[15] An interactive version of this plot is available in the online appendix.

We now seek to provide an answer to **RQ4**. By doing so, we aim at providing a guideline for developers to follow when aiming at reducing energy consumption of an application, by refactoring energy greedy patterns. In other words, we want to identify which combinations should be avoided/adopted, and how to decide on the potentially ambiguous cases.

We believe our recommendations should be interpreted as a confirmation of the possibility that, for certain combinations of refactorings, the energy savings are not cumulative. We argue that these results present a way of looking at the **most likely** scenario for a combination, when it comes to energy savings.

For the majority of combinations, the results indicate that the gains are consistently higher than the losses (as shown in Figure 2). However, for the ones identified in Figure 3 with having *negative* connections, applying only one refactoring in the same context has proven to result in higher energy savings. A joint and more in-depth analysis of both figures allows us to propose the following guidelines:

$\mathcal{G}_1$: Avoid combinations whose gains are not significantly larger than the losses (Figure 2), **specially when they have at least one *negative* connection** (Figure 3). Looking, for instance, at combination *[#4, #5, #6, #7]*, we see that it has 4 *negative* connections, one for each of its individual refactorings. Additionally, it had more tests resulting in losses, and with a higher impact compared to the the ones resulting in gains. Another example is the combination *[#5, #8]*. We consider these combinations to be **harmful**.

$\mathcal{G}_2$: Adopt combinations that (only) have positive connections. This is the case of 11 combinations, that we argue developers should apply. Not only do their gains surpass their losses, but there is also statistical evidence that the gains are expected to increase if all refactorings in the combination are applied, instead of applying only one of them. Combination *[#4, #5, #7, #8]* is the one where this is more notorious: all tests resulted in gains, and the combination has 4 *positive effect* connections, with either a large, very large, or huge effect size. We classify these combinations as **effective**.

$\mathcal{G}_3$: Even when considering a combination that resulted in substantially higher gains, it might be preferable to do individual refactorings if the combination (only) has negative connections or no connections at all. Combinations such as *[#3, #4]* resulted in substantially higher gains but have *negative* connections. This means that, while the combination is **helpful**, further analysis is needed to decide on what to refactor; this may involve, e.g., counting the occurrences of EGAPs *#3* and *#4* in a particular application.

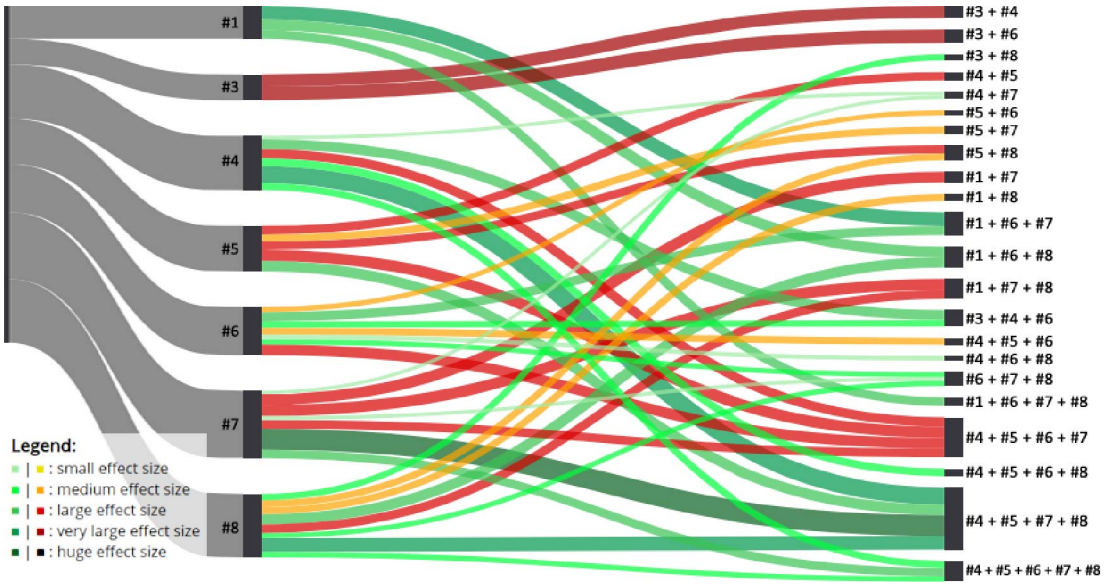

Fig. 3. Gains Comparison: EGAPs vs Combinations

VII. Threats To Validity

We focused on studying the impact of refactoring EGAPs over a large set of applications and on scenarios that try to simulate realistic usage. There are several aspects that may affect the validity of our work and findings.

1. While we have used an experimental setup that is not totally aligned with related works, the fact is that such works have themselves implemented different setups. Also, although some potential measurement noise may exist due to unpredictable actions, such as activity management or garbage collection, there are strong evidences that our conclusions are drawn upon correct data. For once, our results for individual EGAPs (the ones for which a comparison is possible) are well aligned with the findings in the literature. Also, running 25 scenarios per application aids in minimizing the potential error margin. Finally, Trepn is accepted as an effective measurement tool, as it was already used in multiple other energy-related studies.

2. Using random events to define tests may not ensure that applications have all their features explored. However, this is the only generic way to simulate realistic, context-free, usage scenarios, which are essential within our methodology.

3. Our conclusions are sustained on the gains/losses obtained for each refactoring or combination. Extensive as our work may be, this might not reflect all possible scenarios Nevertheless, since our methodology ensures that any significant gain/loss is due to the performed refactorings only, and we test a very large number of applications 25 times without enforcing the use of refactored code, we argue that our study provides a compelling evidence for the most likely scenarios.

VIII. Conclusions And Future Work

This paper performed an analysis over a large-scale repository of Android applications, using as subjects, individually and combined, 11 code patterns termed EGAPs, which were individually studied in previous research to understand how replacing them can influence energy consumption. Our analysis considered the global impact that refactoring has on an application, which also differs from previous approaches.

Our work provides several findings that can guide developers in improving the energy efficiency of their code. For once, refactoring individual EGAPs consistently leads to energy savings, which is aligned with previous findings. Refactoring combinations of EGAPs, however, can sometimes produce less gains than expected, so the decision on whether or not to refactor a certain combination needs to be properly evaluated. The analysis is fully automated, and was achieved by open-source tools, which are publicly available for others to use.

We believe our work raises a relevant direction for future research. Although our extensive and in-depth study has provided experimental and statistical evidence that combining as much refactorings as possible may sometimes not produce the optimal energy efficiency, it would be interesting to further understand, when it occurs, *why* this is the case. Nevertheless, we strongly believe this direction should be addressed by a dedicated study, that can build on our (novel) findings.

Acknowledgments

We would like to thank Cristina Videira Lopes (Univ. of California Irvine) for helping us obtain the Android repository, Jácome Cunha (Univ. of Minho) for the helpful discussions, and the anonymous reviewers for their valuable feedback. The first author is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project UID/EEA/50014/2019, and also by FCT grant SFRH/BD/132485/2017. The second author is financed by the ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalization - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718.

REFERENCES

[1] "The most wanted smartphone features," https://www.statista.com/chart/5995/the-most-wanted-smartphone-features, accessed: 2018-01-24.

[2] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What Do Mobile App Users Complain About?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, May 2015.

[3] "Our phones and gadgets are now endangering the planet," https://www.theguardian.com/commentisfree/2018/jul/17/internet-climate-carbon-footprint-data-centres, accessed: 2018-01-24.

[4] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, January 2019.

[5] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "EARMO: An Energy-Aware Refactoring Approach for Mobile Apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, Dec 2018.

[6] L. Cruz and R. Abreu, "Using Automatic Refactoring to Improve Energy Efficiency of Android Apps," *CoRR*, vol. abs/1803.05889, 2018.

[7] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of Android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 115–126.

[8] L. Cruz and R. Abreu, "Performance-based Guidelines for Energy Efficient Mobile Applications," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17. IEEE Press, 2017, pp. 46–57.

[9] H. Jiang, H. Yang, S. Qin, Z. Su, J. Zhang, and J. Yan, "Detecting Energy Bugs in Android Apps Using Static Analysis," in *Formal Methods and Software Engineering*, Z. Duan and L. Ong, Eds. Springer International Publishing, 2017, pp. 192–208.

[10] D. Li and W. G. J. Halfond, "An Investigation into Energy-saving Programming Practices for Android Smartphone App Development," in *Proc. of 3rd Int. Workshop on Green and Sustainable Software*, ser. GREENS 2014. ACM, 2014, pp. 46–53.

[11] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards Verifying Android Apps for the Absence of No-sleep Energy Bugs," in *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, ser. HotPower'12. USENIX Association, 2012.

[12] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study," in *Proc. of 11th Working Conf. on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 2–11.

[13] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated Energy Optimization of HTTP Requests for Mobile Applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 249–260.

[14] G. Pinto, F. Castor, and Y. D. Liu, "Mining Questions About Software Energy Consumption," in *Proc. of 11th Working Conf. on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 22–31.

[15] G. Pinto and F. Castor, "Characterizing the energy efficiency of java's thread-safe collections in a multi-core environment," in *Proc. of SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS*, vol. 14, 2014.

[16] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language," in *2016 IEEE 23rd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 517–528.

[17] R. Pereira, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "The Influence of the Java Collection Framework on Overall Energy Consumption," in *Proc. of 5th Int. Workshop on Green and Sustainable Software*, ser. GREENS '16. ACM, 2016, pp. 15–21.

[18] R. Pereira, P. Simão, J. Cunha, and J. a. Saraiva, "jStanley: Placing a Green Thumb on Java Collections," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. ACM, 2018, pp. 856–859.

[19] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva, "Towards a Green Ranking for Programming Languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP 2017. ACM, 2017, pp. 7:1–7:8.

[20] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. ACM, 2017, pp. 256–267.

[21] S. Nakajima, "Model-based Power Consumption Analysis of Smartphone Applications," in *16th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29th, 2013.*, 2013.

[22] S. Nakajima, "Using Real-Time Maude to Model Check Energy Consumption Behavior," in *FM 2015: Formal Methods*, ser. LNCS, N. Bjørner and F. de Boer, Eds. Springer Int. Publishing, 2015, vol. 9109, pp. 378–394.

[23] M. Couto, P. Borba, J. Cunha, J. P. Fernandes, R. Pereira, and J. Saraiva, "Products Go Green: Worst-Case Energy Consumption in Software Product Lines," in *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, ser. SPLC '17. ACM, 2017, pp. 84–93.

[24] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "Helping Programmers Improve the Energy Efficiency of Source Code," in *Proc. of the 39th International Conference on Soft. Eng. Companion*, ser. ICSE-C 2017. ACM, 2017, pp. 238–240.

[25] R. Rua, M. Couto, A. Pinto, J. Cunha, and J. Saraiva, "Towards using Memoization for Saving Energy in Android," in *Proceedings of the XXII Iberoamerican Conference on Software Engineering*, ser. CIbSE, 2019, pp. 279–292.

[26] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *Green and Sustainable Software (GREENS), 2012 First Int. Workshop on*. IEEE, 2012, pp. 55–61.

[27] C. Sahin, L. Pollock, and J. Clause, "How Do Code Refactorings Affect Energy Usage?" in *Proc. of 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. ACM, 2014, pp. 36:1–36:10.

[28] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, "Empirical Evaluation of the Energy Impact of Refactoring Code Smells," in *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability*, ser. EPiC Series in Computing, B. Penzenstadler, S. Easterbrook, C. Venters, and S. I. Ahmed, Eds., vol. 52, 2018, pp. 365–383.

[29] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond, "Integrated Energy-directed Test Suite Optimization," in *Proc. of 2014 Int. Symposium on Software Testing and Analysis*, ser. ISSTA 2014. ACM, 2014, pp. 339–350.

[30] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proc. of Eighth Int. Conf. on Hardware/Software Codesign and System Synthesis*, ser. CODES/ISSS '10. ACM, 2010, pp. 105–114.

[31] 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 *Proc. of 7th ACM European Conf. on Computer Systems*, ser. EuroSys '12. ACM, 2012, pp. 29–42.

[32] S. Hao, D. Li, W. Halfond, and R. Govindan, "Estimating Android applications' CPU energy usage via bytecode profiling," in *Green and Sustainable Software (GREENS), 2012 First Int. Workshop on*, June 2012, pp. 1–7.

[33] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating Source Line Level Energy Information for Android Applications," in *Proc. of 2013 Int. Symposium on Software Testing and Analysis*, ser. ISSTA 2013. ACM, 2013, pp. 78–89.

[34] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 39:1–39:40, 2015.

[35] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "EcoDroid: An Approach for Energy-based Ranking of Android Apps," in *Proc. of 4th Int. Workshop on Green and Sustainable Software*, ser. GREENS '15. IEEE Press, 2015, pp. 8–14.

[36] N. Hegde, E. L. Melanson, and E. Sazonov, "Development of a real time activity monitoring Android application utilizing SmartStep," in *Proceedings of the 2016 IEEE 38th Annual International Conference of the Engineering in Medicine and Biology Society (EMBC)*, 2016, pp. 1886–1889.

[37] Y. Hu, J. Yan, D. Yan, Q. Lu, and J. Yan, "Lightweight energy consumption analysis and prediction for Android applications," *Science of Computer Programming*, pp. 132–147, 2018, special Issue on TASE 2016.

[38] M. Couto, C. T., J. Cunha, J. P. Fernandes, and J. Saraiva, "Detecting Anomalous Energy Consumption in Android Applications," in *Programming Languages*, ser. LNCS, F. M. Quintão Pereira, Ed. Springer Int. Publishing, 2014, vol. 8771, pp. 77–91.

[39] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2209–2235, Aug 2019.

[40] L. Cruz, R. Abreu, J. Grundy, L. Li, and X. Xia, "Do energy-oriented changes hinder maintainability?" 2019.

[41] R. Saborido, R. Morales, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "Getting the most from map data structures in Android," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2829–2864, 2018.

[42] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of Android-specific code smells: Tshe aDoctor project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 487–491.

[43] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "UCI Source Code Data Sets," 2010. [Online]. Available: http://www.ics.uci.edu/~lopes/datasets/

[44] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "DéJàVu: A Map of Code Duplicates on GitHub," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 84:1–84:28, 2017.

[45] L. Cruz, R. Abreu, and J.-N. Rouvignac, "Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring," in *IEEE/ACM International Conference on Mobile Software Engineering and Systems, MobileSoft 2017*, ser. MOBILESoft '17, 2017, pp. 205–206.

[46] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy Profiles of Java Collections Classes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 225–236.

[47] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet? (E)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. IEEE Computer Society, 2015, pp. 429–440.

[48] S. Sawilowsky, "New Effect Size Rules of Thumb," *Journal of Modern Applied Statistical Methods*, vol. 8, pp. 597–599, 11 2009.