# E-GreenDroid: Effective Energy Inefficiency Analysis for Android Applications

Jue Wang[§†], Yepang Liu[‡], Chang Xu[§†], Xiaoxing Ma[§†], and Jian Lu[§†]
[†]Dept. of Computer Sci. and Tech., Nanjing Univ., Nanjing, China
[§]State Key Lab for Novel Software Tech., Nanjing Univ., Nanjing, China
[‡]Dept. of Computer Sci. and Engr., The Hong Kong Univ. of Sci. and Tech., Hong Kong, China
[§†]juewang591@gmail.com, [‡]andrewust@cse.ust.hk, [§†]{changxu*, xxm, lj}@nju.edu.cn

## ABSTRACT

Energy inefficiency of smartphone apps is one of the important non-functional issues. It is common, but difficult to diagnose, and often involves sensor usage. GreenDroid provides a novel approach to systematically diagnose energy inefficiency problems in smartphone apps running on Android platforms. It derives an application execution model (AEM) from Android framework and leverages it to realistically simulate an application's runtime behaviors. It also automatically analyzes an application's sensory data utilization, monitors sensor listener and wake lock usage, and reports actionable information to developers.

However, GreenDroid has several limitations. First, other than Android 2.3, it does not support other newer versions of Android. Second, GreenDroid doesn't provide an actionable and reusable state machine based on AEM. Third, its implementation and report generation need optimization. This work focuses on extending GreenDroid's functionality of diagnosing energy inefficiency problems in Android apps. We re-implement GreenDroid on the newest version of Java Pathfinder(JPF), update and optimize the execution simulation process as well as library modeling. Besides, this work adds support to new Android features such as Fragment, and abstracts a separate and reusable state machine out of AEM. With our evaluation, we demonstrate that the extended GreenDroid (E-GreenDroid) can analyze those apps with new Android features while being the same effective as the original version.

## CCS Concepts

•**Software and its engineering** → **Software performance; Software testing and debugging;**

## Keywords

energy inefficiency; smartphone application; sensory data utilization

*Corresponding author.



*Internetware '16, September 18 2016, Beijing, China*



DOI: http://dx.doi.org/10.1145/2993717.2993720

## 1. INTRODUCTION

In recent years smartphone and its apps develop rapidly. Users often want their smartphones to stay on as long as possible with their limited batteries. Meanwhile, many apps involve sensor usage. These apps use sensors to provide context-aware services. However, sensor usage can be energy-consuming if not used cost-effectively [17]. Thus, energy efficiency has become an important non-functional issue to consider in smartphone application development.

However, our investigation [9] shows that 33 of 174 popular Android apps we investigated have received strong complaints from users for energy inefficient problems. Many of these problems are due to sensor usage. This is because Android framework let developers manage sensors [2], and developers often overlook energy inefficiency problems.

Locating energy inefficiency problems is the first step to fix them, but it's rather difficult, because the problems may only appear in a few execution states and it requires labor-intensive efforts to identify these states. However, according to our findings in [9], there are two common patterns of coding that may be the sign of energy inefficient problems.

**Sensor listener and wake lock misusage.** Every application needs to register listeners and specify sensing rate for each sensor to get information, and the sensors won't stop feeding data to their listeners as long as the listeners haven't been unregistered. Therefore, forgetting to properly unregister listeners can cause energy waste. Similarly, wake lock is acquired by an application when it needs to perform long-running computation. If the wake lock isn't properly released, the phone will stay on for long time after the computation is done and thus it causes energy waste.

**Sensory data underutilization.** Sensory data is fed by sensors with cost of energy, and thus should be used effectively. If the usage of sensory data isn't worth the cost of energy, then it's underutilized and this can be a sign of energy inefficiency problems.

Based on these patterns, GreenDroid aims to automatically diagnose Android apps and identify the appearances of these patterns [9]. It simulates runtime behaviors of an application, and monitors the sensor listeners registration/ungistration ,wake lock acquirement/release as well as the sensory data usage. GreenDroid is implemented on top of Java PathFinder (JPF) [21]. It has two major components, *Runtime Controller* and *Sensory Data Utilization Analyzer*, which simulates runtime behaviors and monitors sensory data usage, respectively.

The approach within GreenDroid proves to be novel and effective [9], but its implementation requires update and

optimization. The original GreenDroid supports Android 2.3. Now most of the apps use features of newer versions of Android. Thus the original GreenDroid cannot conduct effective analysis on these apps, which leads to reduction of practicality, and this reduces the possibilities to reuse this implementation in other research areas. The other problem is that it lacks of an abstracted state machine for runtime behavior simulation. Our approach derives a model to guide the simulated execution of Android apps. Original GreenDroid plants codes involving this model across its program codes, which makes it difficult to manage, debug, update and extend. An abstracted state machine of this model can also be reused for other researches involving Android application execution. Finally, GreenDroid is designed to generate actionable report to developers, but it can be better organized to accent more important information. It lacks of an overall result of the analysis, which is expected at the beginning of a report. Moreover, it lacks detailed information, which may help developers fix detected problems. For instance, it doesn't give information about sensory data used during the execution.

Therefore, we focus on updating and optimizing GreenDroid's implementation, as well as proposing an abstracted and reusable state machine for Android application execution. We extend GreenDroid to support features of Android 5.0, including new APIs and components. We also propose a state machine based on GreenDroid's *Application Execution Model* (AEM), which is reusable for any Android application analysis. At the same time, we better organize the report that is generated after analyzing an application, making it accent more important information and more readable to developers. We refer to our extended GreenDroid as E-GreenDroid.

To evaluate the effectiveness of E-GreenDroid, we use popular Android apps with features of Android 5.0 as test subjects. We use both versions of GreenDroid to analyze these apps and compare results. We also use test subjects that were selected to evaluate GreenDroid [9] to study whether E-GreenDroid is as effective as the original GreenDroid. The results show E-GreenDroid can support new features and maintain effective.

In summary, we make the following contributions in this paper:

- We extend GreenDroid to support features of Android 5.0 and optimize its implementation, including library model updating, execution model updating, simulation optimization, etc. We also better organize the report, making it more readable to developers.
- We abstract a state machine for Android application execution from GreenDroid's AEM. The state machine can guide application execution and it's reusable for any analysis concerning execution of Android apps.
- We evaluate E-GreenDroid with both the popular Android apps with new features and the apps used to evaluate original GreenDroid. E-GreenDroid successfully located real energy inefficiency problems in all the apps, suggesting its effectiveness.

The rest of this paper is organized as follows. Section 2 introduces the basics of Android apps and gives an motivating example for both GreenDroid and our extension. Section 3 gives the introduction of the approach within GreenDroid.

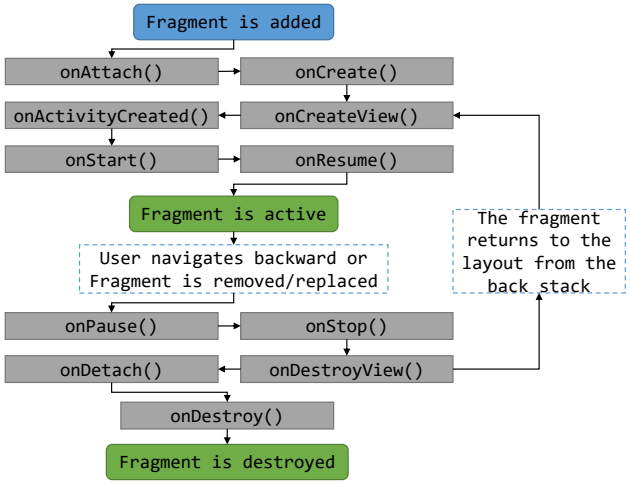



Figure 1: The life cycle of a Fragment

Section 4 presents the extension of GreenDroid. Section 5 evaluates E-GreenDroid and discusses the experimental results. Section 6 reviews related work and finally Section 7 concludes this paper.

# 2. BACKGROUND AND MOTIVATION

## 2.1 Backgroud

Java apps run on Android platform, and these apps contain four types of components:

**Activity.** An application can comprise multiple Activities. Only Activities contain graphical user interfaces.

**Broadcast receiver.** Broadcast receivers receive system-wide broadcasted messages and respond to them accordingly.

**Service.** Services conduct long-running tasks in the background. They are often started by Activities.

**Content provider.** Content providers provide an interface for querying or modifying shared application data.

These four major components can comprise many other components. Fragment is one of these components and it's used as an example for introduction of our extra feature support in Section 4.2.1. So we introduce it here.

**Fragment.** Fragments can be regarded as pieces of an Activity's user interface or/and behaviors of it[1]. An Activity can have many different Fragments and can change the current active Fragment at runtime through FragmentManager. Some of an Activity's GUI components and execution logic are contained within its each Fragment. Figure 1 shows a Fragment's life cycle. A Fragment's life cycle always binds with its belonged Activity.

## 2.2 Motivating Example

Here, we present a real energy inefficiency problem found in LocWriter2 [4] that has been confirmed by its developers. This example is used as motivating example for both GreenDroid and our extension. Figure 2 gives a simplified version of the concerned code. It has three main parts, a Fragment `MainFragment`, its inner class, a BroadcastReceiver `LocReceiver`, and a Service `LocService`.

From the example we can see that the Fragment binds a `onClickListener` to the Button `start` (Lines 12-19). When it's clicked, it starts `LocService` (Lines 16-18), and `LocService` begins a LocationListener `loclistener` to collect

```
1 public class MainFragment extends Fragment {
2   public View onCreateView(LayoutInflater
3       inflater, ViewGroup container,
4       Bundle savedInstanceState){
5     View view = inflater.inflate(
6     R.layout.frag_main,container,false);
7     LocReceiver locReceiver = new LocReceiver();
8    IntentFilter filter = new IntentFilter();
9    filter.addAction("android.intent.action.Location");
10    getActivity().registerReceiver(locReceiver, filter);
11    Button start=(Button)view.findViewById(R.id.start);
12    start.setOnClickListener(new View.OnClickListener(){
13      @Override
14      public void onClick (View v) {
15        String text =((Button)v).getText().toString();
16        Intent startIntent = new Intent(
17            getActivity(), LocService.class);
18        getActivity().startService(startIntent);
19      }});
20    return view;
21  }
22  public class LocReceiver extends BroadcastReceiver{
23    @Override
24    public void onReceive ( Context context,
25    Intent intent){
26      Location loc = (Location)intent
27         .getExtras().get("location");
28      TextView text =(TextView)getActivity()
29         .findViewById(R.id.text);
30      String lo = Double.toString(loc.getLongitude());
31      String s=lo;text.setText(s);
32    }}}
```

```
33 public class LocService extends Service{
34   LocationManager lm;
35   LocationListener loclistener;
36   @Override
37   public int onStartCommand (Intent intent,
38   int flags, int startId){
39     startLocListener();
40     return super.onStartCommand (
41         intent,flags,startId);
42   }
43   public void startLocListener (){
44     Intent intent;
45     lm = (LocationManager)getSystemService(
46         LOCATION_SERVICE);
47     try{
48       loclistener = new LocationListener(){
49         @Override
50         public void onLocationChanged(
51             Location location){
52           intent = new Intent(
53               "android.intent.action.Location");
54           intent.putExtra("location",location);
55           sendBroadcast(intent);
56         }
57       }
58       lm.requestLocationUpdates(
59       LocationManager.GPS_PROVIDER,0,
60          0,loclistener);
61     }catch(SecurityException e){
62       //nothing
63     }
64   }}
```

Figure 2: Motivating example of LocWriters2's problematic code

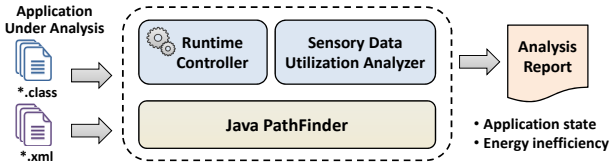



Figure 3: GreenDroid overview

location sensory data (Lines 45-60). When the data indicates that location changes, `loclistener` sends a broadcast message with new sensory data to `LocReciever` (Lines 52-55). Then `LocReceiver` changes the GUI element TextView `text` to show this new location (Lines 26-31).

The whole process seems to be reasonable and efficient. But if LocWriter2's user clicks the `start` Button and then switches to another application, all the GUI elements become invisible. Yet `loclistener` keeps getting sensory data and sending it to `LocReceiver` which will keep updating an invisible GUI element. Thus, the sensory data is underutilized and this brings the energy inefficiency problem.

GreenDroid aims to automatically detect this kind of energy inefficiency problems. But as the example shows, most of the concerned code are within the Fragment, which isn't supported by original GreenDroid. Thus, it cannot detect this problem during analysis. Therefore, we focus on extending GreenDroid's ability to conduct effective analysis.

# 3. INTRODUCTION OF GREENDROID

In this section, we present the introduction of GreenDroid. We first present the overview of GreenDroid (Section 3.1), and then give brief introduction to its two major components (Section 3.2 and Section 3.3).

## 3.1 Overview of GreenDroid

GreenDroid simulates the execution of an Android application, and analyzes utilization of sensory data. It contains an Android application execution model and a tainting-based technique for analyzing sensory data utilization. Figure 3 shows the high-level abstraction of GreenDroid [9]. It takes Java bytecode and necessary configuration files of an application as input, and shows a detailed report as output. The Java bytecode can be obtained by compiling its source code or transforming its Dalvik bytecode [15]. The configuration files specify the application's components, GUI layouts, etc.

The general idea is that we use JPF's Java virtual machine (JVM) to execute an Android application in order to systematically explore the application's state space. The *Runtime Controller* generates input events and guides the execution of the application for state space exploration. The *Sensory Data Utilization Analyzer* analyzes the application's utilization of sensory data at each explored state. We present how to guide the execution and how to analyze sensory data utilization below.

## 3.2 Runtime Controller

An Android application starts when its main Activity is created, and ends when all of its components are destroyed [9]. Its logic is specified in a set of loosely coupled event handlers. An Android application takes different events as input and calls handlers to handle these events. The handlers are implicitly called at runtime. Each call of a handler may lead to state change of the application by modifying its components' data. Thus, an *application state* can be represented by a sequence of event handlers which have been called. Now the problems are how to generate proper events as input and how to schedule handler for each event. We elaborate them below.

To generate proper events, GreenDroid analyzes the configuration files of an application to gain information about its GUI components and Activities. It specifies a set of events for each GUI component. Besides, GreenDroid analyzes the configuration files to specify a set of possible system events. With all these sets of events, GreenDroid generates a set of possible events for each Activity. During the execution, GreenDroid takes one of the possible events of the application's current active Activity and uses it as input.

We can generate all possible event sequences for a limited length by repeating this process[1].

To properly schedule handlers at run time, we derive an *Application Execution Model* (or AEM) from Android specifications [9], and leverage it to guide the scheduling. It's a collection of temporal rules that are enforced at runtime(unary temporal connective $\Box$ means "always"):

$$AEM := \Box \bigwedge_i R_i$$

Each temporal rule is expressed in the following form:

$$R_i := [\psi], [\phi] \Rightarrow \lambda$$

$\psi$ and $\lambda$ are both temporal formulae. They are expressed in linear-time temporal logic, and they refer to what has happened in an execution and what should be done in the future, respectively. $\phi$ is a propositional logic formula evaluating what event is received. In summary, the rule means that $\lambda$ should be executed if $\psi$ and $\phi$ both hold. The details of the model can be found in [9].

As such, *Runtime Controller* can guide the execution of an Android application to explore its states systematically.

## 3.3 Sensory Data Utilization Analyzer

During the execution, GreenDroid analyzes the utilization of sensory data. Sensory data will be transformed into different forms and will be consumed at different states. In order to track its flow and to analyze its utilization, dynamic tainting is required [8]. There are three phases for the technique [9]: (1) tainting each sensory datum with an unique mark, (2) propagating taint marks as the application executes, and (3) analyzing sensory data utilization at different states during the execution.

Tainting the datum is trivial. GreenDroid uses mock sensory data from existing data pool, and the datum can be modified at will. To propagate taint marks, GreenDroid does so on bytecode level. It has a collection of rules for each bytecode instruction about how to propagate the marks [9]. By this, the usage of sensory data can be traced.

To Analyze sensory data utilization, we define the metric of *data utilization coefficient* (DUC) by Equation (1) [9]:

$$DUC(s,d) = \frac{usage(s,d)}{Max_{s' \in S, d' \in D}(usage(s',d'))} \quad (1)$$

The DUC of sensory data $d$ at state $s$ is defined as the ratio between usage of $d$ at $s$ and the maximum usage of any sensory data at any state [9]. This indicates that low DUC suggests low utilization of sensory data.

For usage, it's defined as such:

$$usage(s,d) = \sum_{i \in Instr(s,d)} weight(i,s) \times rel(i) \quad (2)$$

$Instr(s,d)$ is the set of bytecode instructions executed after sensory data $d$ are fed. Whether the bytecode instruction $i$ uses the sensory data $d$ is represented by $rel(i)$. And according to whether it brings benefits to users like changing visible GUI elements or saving data to database, the function $weight(i,s)$ gives the proper weight to instruction $i$.

As such, GreenDroid is able to analyze the utilization of sensory data and identify the states that sensory data is underutilized.

[1]The length of generated event sequences must be limited, or there will be infinite number of sequences.

At the same time, *Sensory Data Utilization Analyzer* monitors registration/unregistration of sensor listeners and acquirement/release of wake locks. If misusage behaviors of these actions appear, it records and reports them.

# 4. EXTENSION OF GREENDROID

In this section, we present our work of E-GreenDroid. We first show the shortcomings of the original GreenDroid's implementation (Section 4.1), and then present the two parts of our extension, update and optimization (Section 4.2), followed by introduction of our abstracted state machine (Section 4.3). And in Section 4.4 we show the better organized report of E-GreenDroid.

## 4.1 Motivation

The original GreenDroid concerns Android 2.3. Now Android 4.4 is the most popular platform and Android 6.0 has come. The original GreenDroid cannot support many features that now are being used by many apps, which may lead to false analysis results. For example, in Section 2.2 the motivating example has its problematic part of the program in a Fragment. It was introduced in Android 3.0, and thus the original GreenDroid cannot conduct effective analysis on apps using this feature.

Besides, in order to keep the data flow inside the context of the program to trace sensory data, GreenDroid uses stubs and mock classes for execution as library classes. These stubs and mock classes are used as library classes to complete the execution of different components of an application so that the data flow is kept inside the context of the program. As well as original GreenDroid itself, these classes are designed for Android 2.3, and the library modeling of the original GreenDroid is neither precise nor complete, which makes it fail to conduct effective analysis on many apps due to the imprecise modeling of library or the lack of support of Android's new features. Therefore, many modeling classes need to be updated and many new ones should be added. The AEM, as mentioned in Section 3.2, needs to be updated as well.

Finally, some mechanisms of GreenDroid could be optimized to reduce performance overhead and to make GreenDroid more user-friendly, which will be elaborated below. our E-GreenDroid supports new features of Android 5.0, because it is the newest version of Android when we implemented E-GreenDroid and it includes all the features of earlier versions of Android, including the most popular version of the platforms, Android 4.4.

## 4.2 Updates and Optimizations

Here we describe the update and optimization of GreenDroid upon newest version of JPF, which is 1.8. Table 1 shows the list of our update and optimization.

### *4.2.1 Updates*

**Library modeling update.** In order to support new APIs that come with Android 5.0 and to better model Android libraries, we update GreenDroid's library modeling. we update argument format of APIs and their execution processes, while maintaining the traceability of sensory data inside the APIs. Moreover, many new stubs and mock classes are implemented in order to support new library APIs, such as LayoutInflater. Another thing is that since Android 2.3, Android platform tends to use its own data structures, like

Table 1: List of update and Optimization

| Category | Content | Description |
|---|---|---|
| Update | Library modeling update | To better model Android library and to support new features of Android 5.0 |
| | Extra feature support | More sophisticated approaches to support Android 5.0 |
| Optimization | State space reduction | Cut off unnecessary states to reduce performance overhead |
| | Heuristic assignment | Adopt heuristic methods to assign values to sensory datum |

SparseArray which is more suitable for Android platform than HashMap. In order to simulate the real execution process of the application, we also add mock classes for these data structures, modifying their inner structures for E-GreenDroid to trace possible sensory data while maintaining their effectiveness.

**Extra feature support.** Some new features can be supported with new mock classes, while others require more sophisticated approaches. We adopt approaches to support components including Fragment, ToolBar, Spinner, etc. We take Fragment as an example to elaborate how we handle these components' support.

As Section 2.1 says, a Fragment contains part of GUI elements of an Activity as well as these elements' related program logic. A Fragment often includes parts that concern energy inefficiency problems. The original GreenDroid cannot support Fragment's features, so it often fails to effectively analyze apps using Fragment. Therefore we extend it to support Fragment.

In order to properly schedule the handlers of Fragment, we add rules similar to those of Activity into AEM. In general, the rules concern the current states of a Fragment and its belonged Activity. Table 2 shows some examples of the temporal rules for Fragment. The form follows the one mentioned in Section 3.2. Symbol $\odot$ means "previously", `getActivity()` returns the Activity that owns `fra`. and `isCurrentFragment()` determines if `fra` is the current active Fragment. Note that at one time a Activity can only have one Fragment being active. Which Fragment is active is decided by the Activity's program logic.

As mentioned earlier, some GUI components may be contained in a Fragment, so some events will be handled within the Fragment. Therefore, the set of possible events of current Activity has two subsets, the one handled by Activity's logic and the one handled by its current active Fragment's. Each time the current active Fragment changes, the set of possible events also changes. Thus, we maintain a event pool for each Activity and its each Fragment. When an Activity is currently active, we put events in its event pool and events in its current active Fragment's event pool, if any, into the set of possible events, and take them out if current Activity or Fragment changes. In this way we assure that an inactive Fragment's events, even the Fragment's belonged Activity is active, won't be used as input.

One of the reasons Fragment is used is that it can be replaced at runtime. Such actions can involve sensory data utilization, and they do provide benefits to users. Therefore our E-GreenDroid supports such actions. We capture each replacement of Fragment, and switch the current active Fragment. If a Fragment transaction (e.g., replacement) will be added to the backstack, i.e., when **Back** button is clicked after the transaction it will be undone, we create a new copy of current Activity and push it onto the task stack. Then we change the current active Fragment, and update the set of possible events. If this instruction involves sensory data utilization, the proper *weight* will be given. As such, E-GreenDroid is able to support the features of Fragment, and can effectively analyze apps using these features.

### 4.2.2 *Optimizations*

Besides the update, some optimization are made for GreenDroid's execution.

**State space reduction.** when a listener for a sensor is registered, the original GreenDroid takes a sensory datum from existing sensory data pool with random accuracy and values, and feeds it to the application before and after every state change. But some state changes don't concern the sensory data utilization. And in this way the execution processes two different sensory datums before certain state changes happen, which leads to performance overhead. Thus, we optimize the data feeding policy to cut off unnecessary sensory data feeding while maintaining the effectiveness of the analysis.

**Heuristic assignment.** Rather than pure random, heuristic methods may be adopted to assign values of sensory data, like assigning values indicating that user keeps going in the same direction for some specific states. These heuristic methods are optional and can be chosen differently for different test subjects.

## 4.3 Reusable State Machine

Here we present the state machine abstracted from AEM. The original GreenDroid doesn't form one state machine to schedule all the event handlers. Instead, it plants partial codes around the whole program project, making it impossible to reuse and difficult to manage.

The state machine we abstract is a black box. it takes an Activity and an event as input, and returns the proper handler as output. For events used for input, we use the possible events that are generated from *Runtime Controller* as mentioned in Section 3.2. The state machine identifies these events and schedules proper handlers. Since different Activities can be at different states and thus for the same event they may need different handlers, the state machine keeps track of each Activity's execution trace and its current state. When an Activity and an event are sent to the state machine, it searches the AEM rules to find a rule that its $\psi$ and $\phi$ match the Activity's execution trace and its current state as well as the event, respectively. If such rule exists, the state machine schedules the handler according to the rule, updates the track of the Activity's execution and state, and returns the handler reference as output. Note that in some cases, the handlers will be called in a row. For example, an Activity's `onCreate()` and `onStart()` handlers are often scheduled in a row. For such occasions, we define a special event, *NOT_ACT_FINISHED_EVENT*. When a handler that can be called in a row is scheduled, E-GreenDroid will see if there are any other events that can be used as input. If not, it keeps using *NOT_ACT_FINISHED_EVENT* as input event to trigger handlers in a row for the Activity, until the Activity can receive real events or it's destroyed.

As such, we abstract a state machine from AEM. Note t-

Table 2: Example temporal rules for Fragment

| |
|---|
| Rule 1:When should the lifecycle event handler fra.onAttach() be called |
| $[\odot fra.getActivity().onCreate()], [\neg ACT\_FINISH\_EVENT \& isCurrentFragment(fra)] \rightarrow fra.onAttach()$ |
| Rule 2:When should the lifecycle event handler fra.onAttach() be called |
| $[\odot fra.onAttach()], [\neg ACT\_FINISH\_EVENT \& isCurrentFragment(fra)] \rightarrow fra.onCreate()$ |
| Rule 3: When should a lifecycle event handler fra.onPause() be called (# 1) |
| $[\odot fra.getActivity().onPause()], [\neg ACT\_FINISH\_EVENT \& isCurrentFragment(fra)] \rightarrow fra.onPause()$ |
| Rule 4: When should a lifecycle event handler fra.onPause() be called (# 2) |
| $[True], [ACT\_FRAGMENT\_REPLACEMENT\_EVENT \& isCurrentFragment(fra)] \rightarrow fra.onPause()$ |

The total numbers of explored states: 9,785
Overall result: *Sensory Data Underutilization*

| DUC Level | Percentage | Priority Level |
|---|---|---|
| 0.00 | 8.30% | Severe |
| 0.33 | 0.31% | Severe |
| 0.67 | 46.46% | Mild |
| 1.00 | 44.94% | Low |

Figure 4: Example of the overview of the analysis

hat given the Activity and events, the state machine can be used in any analysis that involves execution of Android application, thus it's highly reusable.

## 4.4 Report Organization

Here we present the better organized report as output of E-GreenDroid. The original report groups by states that have energy inefficiency problems. It lacks of an overview of the analysis and specific ranks of problem priority. Moreover, though it has reports on *sensory listener and wake lock misusage*, the original report lacks of sufficient information for developers to debug. As such, we organize the report to give an overview of the analysis, as well as providing better organized information for each detected problem.

The reorganized report first gives the overall result of the analysis of sensory data utilization. Figure 4 shows an example, the overview of the analysis of GPSLogger, an application for recording GPS data. The report first presents the total number of explored states and the overall result. In this case it's 9,785 and E-GreenDroid determines that GPSLogger has *sensory data underutilization*. And then it shows how many states (given by percentages) have what levels of DUC. In the example, it shows that 8.30% of the states have DUC of 0.00, which means that these states don't effectively use sensory data at all. Moreover, it shows that 0.31% of the states have DUC of 0.33, 46.46% of the states have DUC of 0.67, and the rest of the states (44.94%) have DUC of 1.00, which means these states have fully usage of sensory data. The report also gives the priority levels of the sensory data underutilization of all the states. According to our earlier GreenDroid paper, a DUC less than 0.5 often suggests severe energy inefficiency problems. Therefore, we define that states with less than 0.50 of DUC have *Severe* sensory data underutilization. Similarly, we define states with DUC between 0.5 and 0.8 have *Mild* sensory data underutilization and those with higher than 0.8 of DUC have only *Low* sensory data underutilization. In this way we define the priority levels of sensory data underutilization.

After the overview, the report shows the details of problematic states. Figure 5 gives an example, a report of one state with *Severe* sensory data underutilization. It first shows the DUC of this state along with its priority level, followed by the APIs that efficiently/inefficiently use sensory data. In this case the DUC is 0, and thus no API efficiently uses the sensory data. At the same time `makeText` and `show` use sensory data inefficiently. At last it shows the full execution trace to reach this state, along with all the values of all the sensory datums fed to the application. Note that in the report, the problematic states' details are ordered by their states' DUC and priority levels.

At last the report shows the detected *sensory listener and wake lock misusage* behaviors. It first shows what kinds of misusage behaviors it detects, and then gives the full execution traces with these misusage behaviors.

# 5. EVALUATION

In this section, we evaluate the effectiveness of our E-GreenDroid. The effectiveness of original GreenDroid itself has been demonstrated [9], so we need to evaluate whether our E-GreenDroid maintain the same effectiveness. Besides, we evaluate whether we truly extend GreenDroid to support features of Android 5.0. As such, we aim to answer following research questions:

- **RQ1 (Effectiveness):** Does E-GreenDroid hold the same effectiveness as the original GreenDroid, i.e., can E-GreenDroid conduct effective analysis on those apps that the original GreenDroid can effectively analyze?

- **RQ2 (Effect of Extension):** Does E-GreenDroid indeed hold effectiveness that the original GreenDroid doesn't, i.e., can E-GreenDroid conduct effective analysis on those apps with features of Android 5.0, which the original GreenDroid cannot?

## 5.1 Experimental Setup and Design

In order to get the answer of **RQ1**, we picked all the Android apps that were used to evaluate the Original GreenDroid [9][11]. 15 open-source Android apps were used as test subjects and 13 of them were detected energy inefficiency problems. We re-complied all the 13 apps on Android 5.0, because Android 5.0 is our target for E-GreenDroid to support. Among the 13 apps 4 of them can no longer run normally due to platform differences and thus they were discarded. We picked the rest 9 apps as test subjects for **RQ1**. Table 3 gives the information about these apps.

In order to get the answer of **RQ2**, we picked four popular apps with new features of Android 5.0 as test subjects. Table 4 gives the information about these apps. GPSLogger-new is an application reconstructed from GPSLogger, a test subject for **RQ1**. the reconstruction doesn't change its functional logic, and therefore should have no effect on analysis results. At the same time, the reconstruction plants new features of Android 5.0 into the application's code, moving all of its GUI elements and behaviors into a Fragment. The other three apps all use new features of Android 5.0. All the apps

Table 3: **RQ1**'s application information and analysis results comparison

| Application | Revisin No. | Lines of code | Availability | Category | Results O/E[1] | Conclusion[2] |
|---|---|---|---|---|---|---|
| Recycle Locator | R-68 | 3,241 | Google Code | Travel&Local | SLM/SLM | same |
| Ushahidi | R-9d0aa75 | 10,186 | GitHub | Communication | SLM/SLM | same |
| AndTweet | V-0.2.4 | 8,908 | Google Code | Social | WLM/WLM | same |
| BableSink | R-d12879a3 | 1,718 | GitHub | Library&Demo | WLM/WLM | same |
| CWAC-Wakeful | R-d984b89 | 896 | GitHub | Education | WLM/WLM | same |
| Sofia Public Transport Nav. | R-114 | 1,443 | Google Code | Transportation | SDU/SDU | same |
| Osmdroid | R-750 | 18,091 | Google Code | Travel&Local | SDU/SDU | same |
| Omidroid | R-863 | 12,427 | Google Code | Productivity | SDU/SDU | same |
| GPSLogger | R-15 | 659 | Google Code | Travel&Local | SDU&SLM/SDU&SLM | same |

[1] Results O means the Qualitative result of the original GreenDroid's analysis, Results E means the Qualitative result of the extended GreenDroid's analysis. We denote **SLM** as *sensor listener misusage*, **WLM** as *wake lock misusage*, and **SDU** as *sensory data underutilization*

[2] Conclusion means whether the two analysis of both versions of GreenDroid give the same Qualitative results.

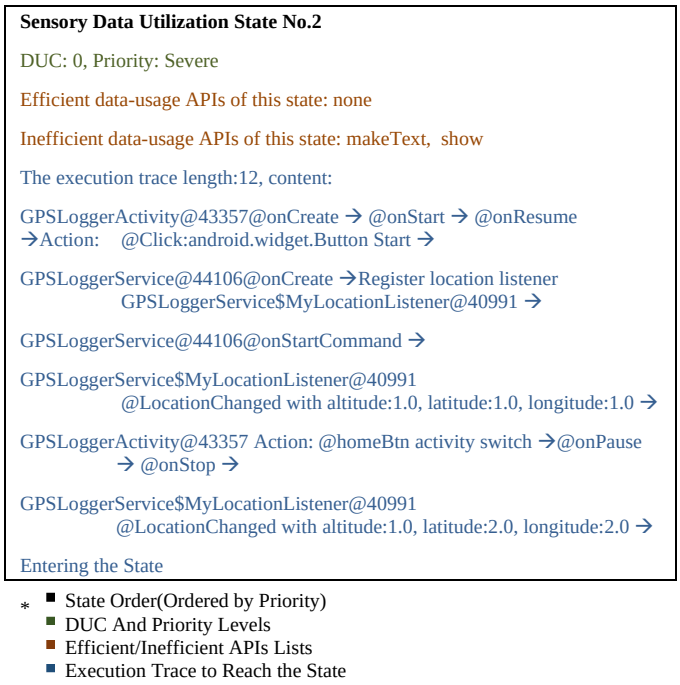



Figure 5: Example of reports about low DUC states

have energy inefficiency problems that have been confirmed by their developers.

We conducted all the experiments on a quadrat-core computer with Intel Core i7 CPU and 8GB RAM, running Windows 10. Further more, we define that an execution is a complete execution process of an Android application for one sequence of user interaction events. We controlled both versions of GreenDroid to generate 5,000 different user interaction event sequences with maximum length of 6 for each test subject. Then we conducted analysis with these events sequences as input. Both versions of GreenDroid use the same sequences for each test subject. This is sufficient for GreenDroid to explore considerable application states.

For both the experiments for **RQ1** and **RQ2**, we run both versions of GreenDroid to conduct analysis on each test subject, and obtain their reports as output. For **RQ1**, we compare same test subject's reports from both versions of GreenDroid. For sensory data Utilization, we further compare the DUC levels across the states from both reports. If the reports both show that the same energy inefficiency problems are detected and the DUC levels across the states from both reports show essentially the same results[2], then we are safe to say that the effectiveness holds for E-GreenDroid. For **RQ2**, we conduct similar experiment. We compare same test subject's reports from both versions of GreenDroid and further compare the DUC levels across the states from both reports. If E-GreenDroid's reports show that the energy inefficiency problems are detected while the original GreenDroid's reports fail to do so, then we are safe to say the extension is effective. The results and discussion of our experiments are shown below.

## 5.2 RQ1: Effectiveness

Table 3 shows the qualitative results of experiments for **RQ1**. For *sensor listener misusage and wake lock misusage*, the reports give explicit results. For sensory data utilization, we define if an application is detected with *Severe* sensory data underutilization, then it has *sensory data underutilization*. From the qualitative results, we can see that both versions of GreenDroid give the same result.

To further demonstrate the effectiveness of our E-GreenDroid, we compare the detailed information of the reports. For *sensor listener misusage and wake lock misusage*, we compare the execution traces of each detected misusage behavior. Through comparison, for each reported misusage behavior, the detailed execution traces are essentially the same, suggesting that for misusage behaviors, E-GreenDroid holds the effectiveness.

For sensory data utilization, we further compare the detailed problematic state information as mentioned in Section 5.1. Figure 6a and 6b shows the DUC level overview for test subject GPSLogger(R-15) from reports of two versions of GreenDroid. From the figure we can see that both versions of GreenDroid report *Severe* sensory data underutilization, and thus the qualitative results are both *sensory data underutilization*. Moreover, the distribution of both charts are essentially the same. The differences appear due to: (1)update of APIs which leads to changes of each API's *weight*, (2)the change of sensory data feeding policy, and (3)the reduction of number of states. These differences don't affect the effectiveness of the results.

For further comparison, for each test subject we analyze and compare two reports's execution traces for each problematic state. The analysis and comparison show that

[2]We use the word *essentially* here and later because the results may not be exactly the same due to the update and extension of GreenDroid as well as the reduction of state space. But as long as the results carry the same messages, they are *essentially* the same and the effectiveness holds.

Table 4: **RQ2**'s application information and analysis results comparison

| Application | Revisin No. | Lines of code | Availability | Category | Results O/E[1] | Conclusion[2] | Cause[3] |
|---|---|---|---|---|---|---|---|
| GPSLogger-new | - | 789 | - | Travel&Local | SDU&SLM /NPD | different | Fragment |
| RedBlackTree | R-0 | 483 | GitHub | Education | WLM/WLM | same | - |
| LocWriter2 | V-0.1.1 | 1,542 | GitHub | Travel&Local | SDU/NPD | different | Fragment &LMU |
| ATT | V0.9-alpha | 52,880 | F-Droid | Navigation | SDU/NPD | different | Fragment &Spinner &LMU |

[1] Results O means the Qualitative result of the original GreenDroid's analysis, Results E means the Qualitative result of the extended GreenDroid's analysis. We denote **NPD** as *no problem detected*, **WLM** as *wake lock misusage*, and **SDU** as *sensory data underutilization*

[2] Conclusion means whether the two analysis of both versions of GreenDroid give the same results.

[3] Cause shows that which improvement of feature support has made E-GreenDroid capable of detecting energy problems in these subjects. We use **LMU** to represent library modeling update, and give specific component names for extra feature support.

both versions of GreenDroid detect the same energy inefficiency problems for each test subject. For example, for GPSLogger(R-15), original GreenDroid detects that 15.69% of states have DUC level of 0.00 while E-GreenDroid detects that 8.30% of states do so. Through comparison we find that even though the percentages and numbers of states are different, the execution traces recorded in both reports suggest the same problems. For instance, both reports of these states give the following same execution trace among others: When sensory data's accuracy is low, the datums will be discarded and no action will be taken, and thus the DUC levels of these states are all 0.00. Normally low accuracy of sensory data lasts for some time and this leads to energy waste. The original GreenDroid reports 1,012 states concerning this problem while E-GreenDroid reports 576 states. This difference is due to the state space reduction. Therefore, though the percentages and numbers of states are not exactly the same in two reports, they are essentially the same.

With these findings, we can answer **RQ1** that E-GreenDroid holds effectiveness.

## 5.3 RQ2: Effect of Extension

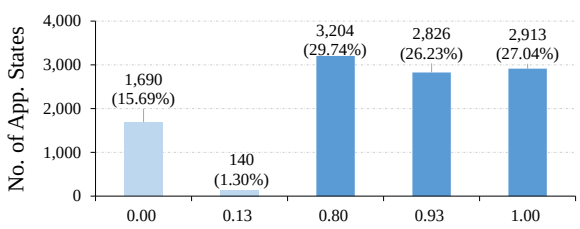
Table 4 presents the qualitative results of experiment for **RQ2**. The definition follows Section 5.2. All the four test subjects have confirmed real energy inefficiency problems. From Table 3 we can see that the original GreenDroid fails to detect any energy inefficiency problem for three test subjects, while E-GreenDroid reports problems in all the test subjects. To show that E-GreenDroid indeed detects real energy inefficiency problems, we further analyze its reports. Figure 6c, 6d and 6e shows the overviews of DUC levels for those test subjects that are reported to have *sensory data underutilization*.

**GPSLogger-new.** As mentioned earlier, GPSLogger-new is a reconstructed application from GPSLogger(R-15). The reconstruction doesn't change its functional logic, and thus all the energy inefficiency problems remain. From Figure 6c we can see that it has exactly the same DUC levels as GPSLogger(R-15), and the execution traces indeed indicate the same problems. GPSLogger-new has all its GUI elements and their program logic within a Fragment, thus the original GreenDroid cannot conduct any effective analysis on it. All of its execution traces only have user events of physical buttons like **Back** or **Home**, and therefore the analysis doesn't give any useful information. On the other hand, E-GreenDroid conducts effective analysis and reports real energy inefficiency problems.

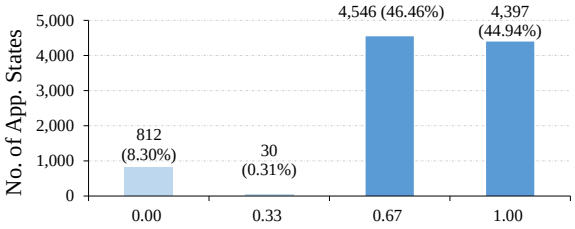
**RedBlackTree.** RedBlackTree is an education application with red-black tree. E-Greendroid reports that in certain cases the application requires wake lock and doesn't releases it when the application terminates, causing the phone to stay active for long time, which leads to energy waste. This has been confirmed by its developer. Both versions of GreenDroid report this problem because the program logic involving the problem doesn't concern new features of Android 5.0. However, the detailed information of original GreenDroid's report shows that it still cannot analyze program logic involving new features of Android 5.0, which may lead to false negative results. Meanwhile, E-GreenDroid conducts full analysis covering all the program logic.

**LocWriter2.** LocWriter2 is an application for recording user's location and presenting records in the screen. As Figure 6d shows, E-GreenDroid reports that it has 33.94% states with DUC level of 0.00, indicating *Severe* sensory data underutilization. Further analysis shows that these states happen when the application starts to record location and user switches to other apps. Since LocWriter2 doesn't write files and only shows records in the screen, the location data is not effectively used at all. For this problem, the listener should be unregistered when the application' GUI turns invisible. This has been confirmed by its developer and our solution has been adopted. LocWriter2's codes about location recording use new APIs of Android 5.0, and some of its GUI elements, e.g., the Button to start the recording, are within a Fragment. Therefore, original GreenDroid cannot conduct any analysis. The analysis process throws exceptions. At the same time, E-GreenDroid's analysis is complete and effective.

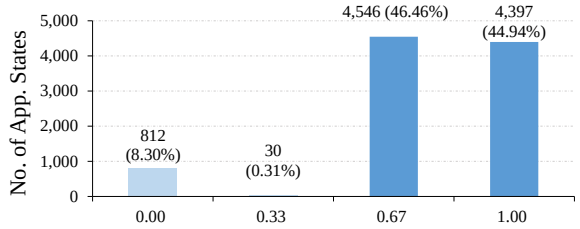
**ATT.** ATT (Android Activity Tracker) is a GPS-tracking application for sports activities. As Figure 6e shows, E-GreenDroid reports 23.94% of states with DUC level of 0.43, indicating *Severe* sensory data underutilization. We find that this also happens when the application starts to record location and user switches to other apps. Different from LocWriter2, ATT stores data to *Shared Preference*, so the sensory data is used with partial effectiveness. And due to the different GUI it has, at certain states the DUC drops below 0.50. This problem is a real energy inefficiency problem and its developer confirms it in its *issue#2* [3]. Besides, ATT uses many APIs that aren't supported by original GreenDroid like Spinner and its GUI elements are all added on-the-fly. Thus, original GreenDroid fails to conduct analysis. At the same time, with all these extra features supported, E-GreenDroid effectively conducts analysis and rep-



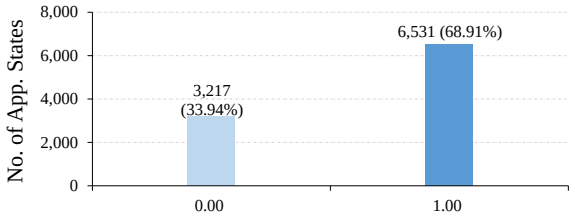

(a) Analysis Result of GPSLogger(R-15) (Original)



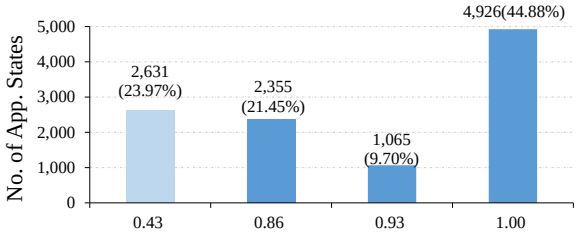

(b) Analysis Result of GPSLogger(R-15) (Extended)




(c) Analysis Result of GPSLogger-new (Extended)




(d) Analysis Result of LocWriter2 (Extended)




(e) Analysis Result of ATT (Extended)

Figure 6: Data utilization results

orts the real energy inefficiency problems, suggesting its effectiveness of analyzing apps with features of Android 5.0.

With these analysis, we can answer **RQ2** that the extension is effective.

## 5.4 Discussion

The approach within GreenDroid is independent of its underlying program analysis framework [9]. The current implementation of E-GreenDroid is still on top of JPF because it's the most suitable framework for our implementation and it's highly extensive. Using JPF may face the limitation of coverage since it's difficult to simulate all the possible interleaving among the user interactions and multiple apps or services. Still, our approach simulates fair amount of interaction sequences and finds real energy inefficiency problems in real world apps, which indicates the effectiveness of our approach and implementation. Nevertheless, we will study the effects of this limitation and overcome it in our future work. Currently, we are publishing a study using white-boxing sampling for self-adaptive covering to partially overcome this limitation.

The extension of adopting to new Android 5.0 is very important in practice. Besides, it makes E-GreenDroid keep up with development of Android, providing its approach and implementation framework the possibilities to be reused in other related research areas, such as consistence checking for Android apps. Besides, the extension provides an abstracted, reusable state machine that can be used in any researches involving Android application execution simulation.

# 6. RELATED WORK

Our work relates to energy efficiency analysis of smartphone apps, Android modeling and JPF extension. We discuss some representative work below.

**Energy efficiency analysis.** Recent years researchers begin to show more and more interest in smartphone application's energy inefficiency problems. Pathak proposed eProf, a fine-grained energy profiler for smartphone to help estimate an application's energy consumption [18]. Oliner et al. proposed a black-box method, Carat, to detect energy anomalies for the whole mobile devices [16]. Cuervoy et al. published a study called MAUI that reduces power needs of Android apps with remote servers [5]. Hasan et al. created detailed profiles of the energy consumed by common operations done on Java collection classes [6]. Zhang et al. published a study on how user choices can affect energy consumption [23]. Our study published in 2015 described how to diagnose energy efficiency and performance for mobile internetware apps [10], while one of our study proposed this year addresses more wake lock problems [12]. GreenDroid shares similar goals, but focuses more on energy inefficiency problems involving sensors by analyzing sensor usage and sensory data utilization.

**Android modeling.** Android modeling includes GUI modeling, execution modeling, etc., and it's vital for effective analysis. Yang et al. proposed a static analysis to create a model of the behaviors of an Android application's GUI [22]. Shye et al. conducted a comprehensive analysis of real smartphone usage, and presented findings on how to model user activities [20]. Preez et al. proposed a study describing how a family of complex system simulation models were developed as a domain related family [19]. Our extension of GreenDroid involves models of execution, GUI and its behaviors, and libraries of Android, but aims to extend the ability of GreenDroid to support features of Android 5.0.

**JPF extension.** JPF is a model checking framework for Java [21]. Mirzaei et al. used JPF to conduct tests of Android apps through symbolic execution [14]. They also proposed a study to automatically generate system input for Android apps using JPF [13]. Heila et al. published a study describing the development of JPF-Android, an Android application verification tool built on JPF [7]. Our extension of GreenDroid naturally extends JPF, since GreenDroid is built on JPF. However, we extend JPF to simulate execution of Android application and analyze sensory data usage during the simulation.

# 7. CONCLUSION

In this paper, we have presented our work of extending GreenDroid's ability to diagnose energy inefficiency problems in Android apps. Our E-GreenDroid can support new features of Android 5.0 such as Fragment, etc. We update its stubs and mock classes as Android library. We also better organize the report to accent important information and abstract a reusable state machine based on AEM. We evaluate it using 13 real Android apps in two experiments. The results show that our extension is effective while our E-GreenDroid holds its original effectiveness.

In the future, we plan to further extend GreenDroid to support more features of Android. We also plan to extend GreenDroid to support concurrency of Android apps. Android apps often have background threads handling long running tasks. Currently, E-GreenDroid cannot support it and simply puts all the execution into one thread. We will extend E-GreenDroid's ability to support concurrency in future.

# 8. ACKNOWLEDGEMENT

This work was supported in part by National Basic Research 973 Program (Grant no. 2015CB352202), and National Natural Science Foundation (Grant no. 61472174, 91318301, 61321491) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

# 9. REFERENCES

[1] Android fragment api. https://developer.android.com/reference/android/app/Fragment.html.

[2] Android sensor management. http://developer.android.com/reference/android/hardware/SensorManager.html.

[3] Att issue#2. https://github.com/bailuk/AAT/issues/2.

[4] Lockwriter2. https://github.com/ArkBriar/LocWriter2.

[5] E. Cuervoy, A. Balasubramanianz, and D. ki Cho. Maui: Making smartphones last longer with code offload. In *MobiSys'10*, 2010.

[6] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. Energy proïňĄles of java collections classes. In *ICSE'16*, 2016.

[7] V. Heila, V. Brink, and W. Visser. Verifying android applications using java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[8] V. Kemerlis, G. Portokalidis, K. Jee, and A. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *VEE'12*, pages 121–132, 2012.

[9] Y. Liu, C. Xu, and S. C. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *PreCom'13*, pages 2–10. IEEE, 2013.

[10] Y. Liu, C. Xu, and S. C. Cheung. Diagnosing energy efficiency and performance for mobile internetware applications. *IEEE Software*, 32(1):67–75, 2015.

[11] Y. Liu, C. Xu, S. C. Cheung, and J. Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *TSE*, 40(9):911–940, 2014.

[12] Y. Liu, C. Xu, S. C. Cheung, and V. Terragni. Understanding and detecting wake lock misuses for android applications. In *FSE'16*, 2016.

[13] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: Automated system input generation for android applications. In *ISSRE'15*, 2015.

[14] N. Mirzaer, S. Malek, C. PĂČsĂČreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2015.

[15] D. Octeau, S. Jha, and P. McDaniel. Retargeting android applications to java bytecode. In *FSE'12*, 2012.

[16] A. Oliner, A. Lyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *SenSys'13*, 2013.

[17] J. Paek, J. Kim, and R. Govindan. Energy-efficient rate-adaptive gps-based positioning for smartphones. In *Mobisys'10*, pages 299–314. ACM, 2010.

[18] A. Pathak, Y. Hu, and M. Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *EuroSys'12*, pages 29–42, 2012.

[19] V. D. Preez, B. Pearce, K. A. Hawick, and T. H. McMullen. Software engineering a family of complex systems simulation model apps on android tablets. In *ICSE'12*, 2012.

[20] A. Shye, B. Scholbrock, G. Memik, and P. Dinda. Characterizing and modeling user activity on smartphones: summary. In *SIGMETRICS'10*, 2010.

[21] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE'00*, pages 3–11, 2000.

[22] S. Yang, H. Zhang, H. Wu, and Y. Wang. Static window transition graphs for android. In *ASE'15*, 2015.

[23] C. Zhang, A. Hindle, and D. M. German. The impact of user chioce on energy consumption. *IEEE Software*, 31(3):69–75, 2014.