2013 IEEE International Conference on Pervasive Computing and Communications (PerCom), San Diego (18--22 March 2013)

Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications

Yepang Liu
Dept. of Comp. Sci. and Engr.
The Hong Kong Univ. of Sci. and Tech.
Hong Kong, China
andrewust@cse.ust.hk

Chang Xu*
State Key Lab for Novel Soft. Tech.
Dept. of Comp. Sci. and Tech.
Nanjing University, Nanjing, China
changxu@nju.edu.cn

S.C. Cheung
Dept. of Comp. Sci. and Engr.
The Hong Kong Univ. of Sci. and Tech.
Hong Kong, China
scc@cse.ust.hk

***Abstract*—Smartphone applications have millions of users. Their energy efficiency is very important. However, we investigated 174 Android applications and found 33 of them suffering serious energy inefficiency problems. Many of these problems are due to ineffective use of sensors and their data. In this paper, we propose a novel approach to systematically diagnose energy inefficiency problems in Android applications. We derive an application execution model from Android specifications, and leverage it to realistically simulate an application's runtime behavior. Our approach can automatically analyze an application's sensory data utilization at different states, and report actionable information to help developers locate energy inefficiency problems and identify their root causes. We built a tool called GreenDroid on top of Java PathFinder and evaluated it using six popularly downloaded Android applications. GreenDroid analyzed these applications in a few minutes, and successfully located real energy inefficiency problems in them.**

Keywords-energy inefficiency; sensory data utilization.

I. INTRODUCTION

Increasing market penetration of smartphones fosters the proliferation of various sensor-based applications. These applications can sense the users' environment and provide context-aware services. For example, Google Maps can navigate users when they hike in a rural area by location sensing. However, limited battery life always restricts such applications' usage as sensing operations are energy-consuming. If sensors are not used cost-effectively, batteries can drain quickly [18]. Energy efficiency thus becomes an important factor to consider in smartphone application development.

Unfortunately, our investigation of 174 popular Android applications shows that 33 of them have received strong complaints from users because of the energy inefficiency problems [8]. Many problems are caused by sensors for two major reasons. First, the Android framework shifts the burden of sensor management to developers [1]. Sensor mismanagement can easily lead to huge energy waste. Second, Android applications are mostly developed by small teams without dedicated quality assurance. Developers tend to focus on functionalities, while largely overlooking energy inefficiency complaints from users, especially when these complaints contain little actionable information.

Locating energy inefficiency problems in Android applications is the first step towards energy consumption optimization. It is difficult because energy inefficiency often occurs only at certain application states. To identify such states, one has to extensively test the application on different devices and perform energy profiling for every state. To figure out the causes, one has to instrument the concerned programs to collect and analyze runtime usage information of sensory data. Such tasks are tedious and require intensive manual efforts. This may explain why only 12 of the 33 energy inefficiency problems found in our investigation have ended up with concrete fixes [8]. Therefore, we aim at an automated energy analysis approach to help developers quickly locate energy inefficiency problems in their Android applications.

Currently there is no well-defined criterion for energy analysis. We closely studied the energy inefficiency problems found in our investigation, and observed that: *Although the root causes of energy waste can be application-specific, they are closely associated with the ineffective use of sensors and their data*. For example, Osmdroid, a popular mapping application, may keep acquiring GPS data to render an invisible map at certain states [17]. Battery energy is thus wasted by location sensing, but the acquired data are not used for users' benefits (rendering an invisible map is meaningless). Our study found two common types of coding phenomena that could lead to energy waste in Android applications.

Sensor listener misusage. To use a sensor, an application needs to register a sensor listener with the Android system, and specify a sensing rate [1]. A listener defines how an application reacts to sensor value or status changes. When the sensor is no longer needed, its corresponding listener should be unregistered. Forgetting to unregister would lead to wasted sensing operations and battery energy [2].

Sensory data underutilization. Sensory data are acquired at the cost of energy, and should be effectively used by an application. Sensory data are "underutilized" when their energy cost outweighs their actual uses. As shown later, sensory data underutilization often suggests design or implementation defects that could cause energy waste.

With these two findings, we propose a novel approach for systematic diagnosis of energy inefficiency problems in Android applications. Our approach conducts code analysis to simulate the runtime behavior of an application. It checks how sensory data are utilized at each explored application state, and monitors sensor listener registration and unregistration operations. We implement our approach on top of Java PathFinder (JPF) [28], a verification framework for Java programs. We will show later that our approach can

*Corresponding author.

analyze location data utilization over 120K states for Osmdroid within three minutes, and successfully locate its energy inefficiency problem. Such efficient and effective energy analysis needs to address two major technical challenges:

Challenge 1. Android applications follow an event-driven programming paradigm. Developers specify the application logic in a set of loosely coupled event handlers that are implicitly called at runtime (see Section II.A). The handler calling order is never specified in code. This causes trouble for existing analysis frameworks like JPF as they rely on explicit calling relationships to analyze programs. Researchers from "Google Summer of Code" project aim to enable JPF to verify Android applications, but have not been very successful over the past several years [9].

Challenge 2. To analyze how sensory data are utilized in an Android application, one needs to know which program data depend on sensory data and how they are used. This requires program instrumentation, which is usually unavailable in source code. Manually making application-specific instrumentation is labor-intensive and error-prone. Hence, it reduces the attraction and practicality of such energy analysis.

To address the first challenge, we derive an application execution model from Android specifications. This model captures application-generic temporal rules that specify calling relationships between event handlers. Enforcing these rules would enable JPF to realistically execute an Android application. To address the second challenge, we monitor an application's execution, and perform dynamic data flow tracking at a bytecode instruction level. By doing so, we can automatically pinpoint those application states where sensory data are underutilized or sensor listeners are misused. In this paper, we make the following contributions:

- We propose a runtime analysis technique to automatically analyze sensory data utilization at different states of an Android application.
- We present an application execution model that captures application-generic temporal rules for event handler scheduling. This model is general enough to be used in other Android application analysis techniques.
- We implement a prototype tool called GreenDroid. To the best of our knowledge, GreenDroid is the first JPF extension that is able to verify Android applications.
- We evaluate GreenDroid using six popular Android applications. GreenDroid successfully located real energy inefficiency problems in four applications, and reported new problems for the remaining two.

The rest of this paper is organized as follows. Section II introduces the basics of Android applications and a motivating example. Section III presents our energy analysis approach. Section IV evaluates our approach and discusses the experimental results. Section V reviews representative related work, and finally Section VI concludes this paper.

II. Background and Motivating Example

A. Background

Android is a popular smartphone platform, where Java applications are programmed in four types of components:

Activity. Activities are the only components that contain graphical user interfaces. An application may comprise multiple activities to provide a cohesive user experience.

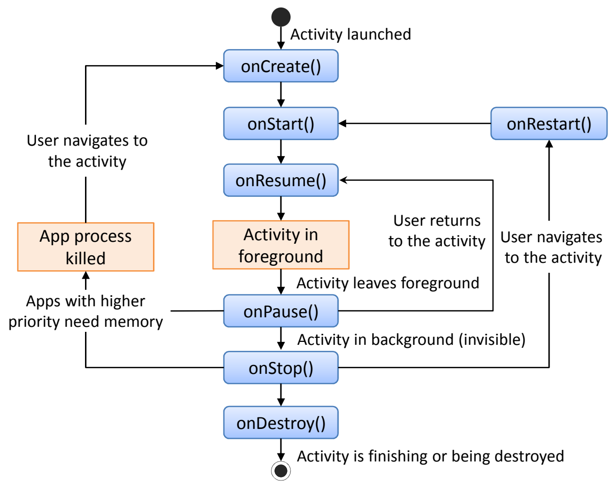
Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
User returns to the activity
User navigates to the activity
App process killed
Activity in foreground
Activity leaves foreground
Apps with higher priority need memory
onPause()
Activity in background (invisible)
onStop()
onDestroy()
Activity is finishing or being destroyed

Figure 1. Activity lifecycle diagram

Service. Services are components that run in the background for conducting long-running tasks like sensor reading. Activities can start and interact with services.

Broadcast receiver. Broadcast receivers define how an application responds to system-wide broadcasted messages. It can be statically registered in an application's configuration file, or dynamically registered at runtime.

Content provider. Content providers manage shared application data, and provide an interface for other components or applications to query or modify these data.

Each application component has a lifecycle defining how it is created, used, and destroyed. Figure 1 shows an activity lifecycle. It starts with a call to onCreate() handler, and ends with a call to onDestroy() handler. An activity's foreground lifetime starts after a call to onResume() handler, and lasts until onPause() handler is called when another activity comes to the foreground. In the foreground, an activity can interact with its user. When it goes to the background and becomes invisible, its onStop() handler would be called. When users navigate back to a paused or stopped activity, the activity's onResume() or onRestart() handler would be called, and the activity would come to the foreground again. In exceptional cases, a paused or stopped activity may be killed for releasing memory to other applications with higher priorities.

B. Motivating Example

Let us discuss a real energy inefficiency problem found in Osmdroid [17]. Figure 2 gives a simplified version of the concerned code. It has three components: (1) MapActivity for displaying a map to its user, (2) GPSService for location sensing and data processing in the background, and (3) a broadcast receiver for handling location change messages (Lines 7–13). When MapActivity is launched, it starts GPSService (Lines 5–6), and registers the broadcast receiver (Lines 15–16). GPSService registers a location listener with the Android system when it starts (Lines 36–47). When the user's location changes, GPSService would process new location data (Line 39), and broadcast a message with the processed data (Lines 41–43). The broadcast receiver would

```
 1.  public class MapActivity extends Activity{
 2.    private Intent gpsIntent;
 3.    private BroadcastReceiver myReceiver;
 4.    public void onCreate(){
 5.      gpsIntent = new Intent(GPSService.class);
 6.      startService(gpsIntent); //start GPSService
 7.      myReceiver = new BroadcastReceiver() {
 8.        public void onReceive(Intent intent) {
 9.          LocData loc = intent.getExtra();
10.          updateMap(loc);
11.          if(trackingModeOn) persistToDatabase(loc);
12.        }
13.      }
14.      //register receiver for handling location change messages
15.      IntentFilter filter = new IntentFilter("loc_change");
16.      registerReceiver(myReceiver, filter);
17.    }
18.    public void onDestroy() {
19.      //stop GPSService and unregister broadcast receiver
20.      stopService(gpsIntent);
21.      unregisterReceiver(myReceiver);
22.    }
23.  }

31.  public class GPSService extends Service{
32.    private LocationManager lm;
33.    private LocationListener gpsListener;
34.    public void onCreate(){
35.      //get a reference to system location manager
36.      lm = getSystemService(LOCATION_SERVICE);
37.      gpsListener = new LocationListener() {
38.        public void onLocationChanged(Location loc) {
39.          LocData formattedLoc = processLocation(loc);
40.          //create and send a location change message
41.          Intent intent = new Intent("loc_change");
42.          intent.putExtra("data", formattedLoc);
43.          sendBroadcast(intent);
44.        }
45.      }
46.      //GPS listener registration
47.      lm.requestLocationUpdates(GPS, 0, 0, gpsListener);
48.    }
49.    public void onDestroy() {
50.      //GPS listener unregistration
51.      lm.removeUpdates(gpsListener);
52.    }
53.  }
```

Figure 2. Motivating example from the Osmdroid application (Issue 53)

then use the new location data to refresh the map (Line 10). If the user has enabled location tracking, these data would also be stored in a database (Line 11). If the Android system plans to destroy MapActivity (Lines 18–22), GPSService would be stopped (Line 20), and both the location listener and broadcast receiver would be unregistered (Lines 21, 51).

If Osmdroid's users switch their smartphones to another application, MapActivity would be put to the background (not destroyed), but GPSService would still keep running for location sensing. If the location tracking functionality is not enabled, all location data would be used to refresh an invisible map. Then, a huge amount of energy would be wasted.

The cause of this energy waste is the delayed unregistration of the location listener. GPS sensing should be disabled if location data are only used to render an invisible map. This resembles a resource leak problem, but existing resource leak detection techniques [3][25] cannot effectively locate energy inefficiency problems for two reasons. First, existing techniques rely on explicit calling relationship to conduct program analysis. Such relationship is not readily available in an Android application's source code. Second, existing techniques can neither distinguish application states nor analyze sensory data utilization. For example, if users have enabled the location tracking functionality before putting MapActivity to the background, then even if the battery is still drained by continuous GPS sensing, we cannot conclude that the energy is wasted (location data are stored for future use) [16]. This motivates us to propose an approach that correlates application states and sensory data utilization to diagnosing energy inefficiency problems in Android applications.

III. Energy Inefficiency Analysis Approach

Our approach contains an Android application execution model and a tainting-based technique for analyzing sensory data utilization. We start with an overview.

A. Approach Overview

Our analysis is based on dynamic information flow [11]. Figure 3 shows its high-level abstraction. It takes as input the Java bytecodes and configuration files of an Android application. The Java bytecodes define the application's program logic, and can be obtained by compiling its source code or transforming its Dalvik bytecodes [15]. The configuration files specify the application's components, GUI layouts and so on. The general idea is that we execute an Android application in JPF's Java virtual machine (JVM)[2], and systematically explore its application states. During an execution, our approach monitors the sensor registration and unregistration operations, and feeds mock sensory data to the application. By tracking the propagation of sensory data as the application executes, we analyze how sensory data are utilized at different application states. Our approach compares sensory data utilization across different states and reports those states where sensory data are underutilized. Our approach also checks which sensor listeners have been forgotten to unregister at the end of an execution, and reports these anomalies.

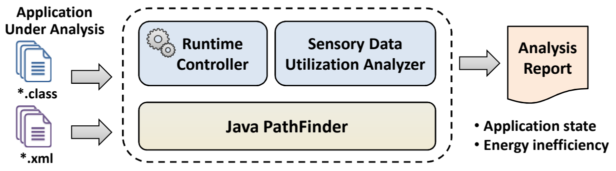

Figure 3. Approach overview

This high-level abstraction looks intuitive, but some challenging questions remain unanswered: How can JPF realistically execute an Android application and enumerate its states? How to identify program data that depend on sensory data? How to measure and compare sensory data utilization at different application states? We answer them below.

B. Application Execution Model

An Android application starts with its main activity, and ends after all its components are destroyed. It keeps handling received events by calling their handlers according to Android specifications. Each call to an event handler may change the application's state by modifying its components' local or global program data. We thus use the sequence of event handlers that have been called to represent an *application state*. To simulate real executions, we need to derive an

[2] On real devices, an Android application runs in a registered-based Dalvik VM, while JPF's JVM is stack-based. This difference does not affect our analysis.

Table 1. Example temporal rules

Rule 1: When should the lifecycle event handler act.onStart() be called?	**Rule 3: When should a dynamic message event handler rcv.onReceive() be called?**
$[\odot act.onCreate()], [\neg ACT\_FINISH\_EVENT] \Rightarrow \bigcirc act.onStart()$	$[\neg \mathrm{rcv}.unreg()\ S_s\ \mathrm{rcv}.reg()], [MSG\_EVENT] \Rightarrow \bigcirc \mathrm{rcv}.onReceive()$
Rule 2: When should a button-click event handler listener.onClick() be called?	**Rule 4: When should a static message event handler Receiver.onReceive() be called?**
$[(\neg act.onPause()\ S_s\ act.onResume()) \wedge (\neg btn.reg(null)\ S_s\ btn.reg(listener))],$ $[BTN\_CLICK\_EVENT] \Rightarrow \bigcirc listener.onClick()$	$[True], [MSG\_EVENT] \Rightarrow \bigcirc Receiver.onReceive()$

application execution model (or AEM) from Android specifications, and leverage it to guide the runtime scheduling of event handlers. Our AEM model is a collection of temporal rules. They are application-generic and should be enforced at runtime (unary temporal connective □ means "always"):

$$AEM \coloneqq \Box \bigwedge_i R_i$$

Each temporal rule is expressed in the following form:

$$R_i \coloneqq [\psi], [\phi] \Longrightarrow \lambda$$

ψ and λ are two temporal formulae expressed in linear-time temporal logic, and refer to the past and future, respectively. ϕ is a propositional logic formula referring to the present. ψ describes what has happened in an execution, ϕ evaluates the current situation (what event is received), and λ describes what should be done in the future. The whole rule means: If both ψ and ϕ hold, λ should be executed next.

We list some temporal rules in Table 1. For the entire collection, readers may refer to our technical report [24]. Propositional connectives $\wedge$, $\Rightarrow$, and $\neg$ in these examples follow their traditional interpretations, and temporal connectives are explained as follows. Unary temporal connective $\bigcirc$ means "next", and its past time analogue $\odot$ means "previously". Binary temporal connective S_s means "strong since". Specifically, a temporal formula "$F_1\ S_s\ F_2$" means that F_2 held at some time in the past, and since then F_1 always holds.

The first rule requires an activity's onStart() handler to be called after its onCreate() handler completes as long as the activity is not forced to finish. The second rule requires a button-click event handler to be called if: (1) the button is clicked, (2) its enclosing activity is at foreground (i.e., the activity's onPause() handler has not been called since the last call to onResume() handler), and (3) its click event listener is registered. The third rule disenables the call to a message event handler before its registration and after its unregistration. The last rule requires that a static message event handler should be called upon any broadcasted message.

Our AEM model is converted to a decision procedure to decide which event handlers to be called according to an application's execution history and its newly received events. Due to page limit, we do not discuss the details in this paper. In addition to the AEM model, our runtime controller also needs to simulate user interactions. It pre-analyzes an application's configuration files to learn an application's GUI layouts. During execution, when the application awaits user interactions, our controller would generate a sequence of interaction events (e.g., click events). We leverage JPF's model checking functionality to systematically generate such sequences to explore different possibilities. Since all event sequences are infinite, we have to limit the number of user interaction events generated in an execution. By this, we execute an application a finite number of times in JPF's virtual machine, and generate distinct sequences of user interaction events for these executions.

C. Sensory Data Utilization Analysis

During program execution, sensory data are transformed and consumed by application components. We track the usage of these data by variables in each bytecode instruction for a precise analysis of sensory data utilization. To do so, we adapt the idea of dynamic tainting [11], and design our tainting-based analysis technique, which contains three phases: (1) taint each sensory datum with a special mark; (2) propagate taint marks as the application executes; (3) analyze sensory data utilization at specific points during the execution. We elaborate on these phases below.

1) Preparing and Tainting Sensory Data

In this phase, mock sensory data from an existing data pool controlled with different precision levels are fed to the application under analysis before and after each user interaction. Each data object representing a sensory datum is tainted with a unique mark before being fed to the application.

2) Propagating Taint Marks

At runtime, each tainted data object is transformed by assignment, arithmetic, relational, or logical operations and control flows. For example, in Figure 2 object *loc* at Line 38 is transformed to another object *formattedLoc* at Line 39, which further affects object *intent* at Lines 41–42; by a message communication, this *intent* object is propagated to a broadcast receiver and converted back to the *loc* object at Line 9, which may or may not affect database contents, depending on variable *trackingModeOn*'s value (Line 11). Such data flows are leveraged to propagate taint marks as well as to identify which program data rely on sensory data.

Our technique propagates taint marks at the bytecode instruction level in JPF's virtual machine. A key advantage of instruction-level taint propagation is that it does not require application-specific program instrumentation, which is both time-consuming and error-prone. Our tainting policy in Table 2 contains 12 propagation rules of the following form:

$$T(A) = T(B) \cup T(C)$$

It means that B's and C's taint marks are merged to be A's taint marks (B and C are optional). Instead of elaborating on each propagation rule, we illustrate taint propagation using a concrete example. The code in Figure 4 uses accelerometer data to detect whether the phone is shuffled (Line 3). If it is, the application's background would be changed (Lines 4–10). The initial taint mark is associated with an object reference *event*. It is propagated to a local variable *values* inside the *isShuffled* method (Rules 8, 5). By assignments, the taint mark is propagated to variables x, y, and z (Rules 3, 4). Next, a local variable *accelerationSquareRoot*

Table 2. Taint propagation policy

Index	Bytecode Instruction	Instruction Semantics	Taint Propagation Rule
1	***Const-op*** *C*	$stack[0] \leftarrow C$	$T(stack[0]) = \varnothing$
2	***Load-op*** *index*	$stack[0] \leftarrow localVar_{index}$	$T(stack[0]) = T(localVar_{index})$
3	***LoadArray-op*** *arrayRef, index*	$stack[0] \leftarrow arrayRef[index]$	$T(stack[0]) = T(arrayRef) \cup T(arrayRef[index])$
4	***Store-op*** *index*	$localVar_{index} \leftarrow stack'[0]$	$T(localVar_{index}) = T(stack'[0])$
5	***StoreArray-op*** *arrayRef, index*	$arrayRef[index] \leftarrow stack'[0]$	$T(arrayRef[index]) = T(stack'[0])$
6	***Binary-op***	$stack[0] \leftarrow stack'[0] \otimes stack'[1]$	$T(stack[0]) = T(stack'[0]) \cup T(stack'[1])$
7	***Unary-op***	$stack[0] \leftarrow \ominus\, stack'[0]$	$T(stack[0]) = T(stack'[0])$
8	***GetField-op*** *index*	$stack[0] \leftarrow stack'[0].instanceField$	$T(stack[0]) = T(stack'[0].instanceField) \cup T(stack'[0])$
9	***GetStatic-op*** *index*	$stack[0] \leftarrow ClassName.staticField$	$T(stack[0]) = T(ClassName.staticField)$
10	***PutField-op*** *index*	$stack'[1].instanceField \leftarrow stack'[0]$	$T(stack'[1].instanceField) = T(stack'[0])$
11	***PutStatic-op*** *index*	$ClassName.staticField \leftarrow stack'[0]$	$T(ClassName.staticField) = T(stack'[0])$
12	***Return-op(non-void)***	$callerStack[0] \leftarrow calleeStack'[0]$	$T(callerStack[0]) = T(calleeStack'[0])$

Index	Detailed Instruction Semantics (*The semantics of the instructions whose index are underlined serve as examples*)
1	Push a constant value *C* onto the operand stack (*stack*[0] represents the value at the stack top after an operation).
2, 3	Load the value of the #*index* local variable onto the operand stack.
4, 5	Pop and store the value at stack top to the #*index* local variable (*stack'*[0] represents the value at the stack top before an operation).
6, 7	Perform the binary operation ⊗ on the two values popped from the operand stack (i.e., *stack'*[0] and *stack'*[1]), and push the result back onto stack.
8, 9	Get a field value of an object on the heap and push the value onto the operand stack. The object reference is popped from the stack (i.e., *stack'*[0]).The object field's name and type can be found by referring to the #*index* slot of the constant pool.
10, 11	Pop and store the value at the stack top (i.e., *stack'*[0]) to an object field on the heap. The object reference is popped from the stack (i.e., *stack'*[1]). The object field's name and type can be found by referring to the #*index* slot of the constant pool.
12	Pop the value at the callee's operand stack top (i.e., *calleeStack'*[0]), and push the value onto the caller's operand stack.

```
public void onSensorChanged(SensorEvent event){
  if(event.sensor.getType() == Sensor.ACCELEROMETER){
    boolean switch = isShuffled(event);
      if(switch){
        if(getBackgroundColor() == RED){
          setBackgroundColor(GREEN);
        } else{
          setBackgroundColor(RED);
        }
      }
   }
}
```

```
public boolean isShuffled(SensorEvent event){
  float[] values = event.values;
  float x = values[0];
  float y = values[1];
  float z = values[2];
  float g = SensorManager.GRAVITY_EARTH;
  float accelerationSquareRoot = (x * x + y * y + z * z) / (g * g);
  if(accelerationSquareRoot >= 2){
     return true;
  }
  return false;
}
```

Figure 4. Example code to demonstrate taint propagation

is tainted accordingly (Rules 6, 4). Finally, the returned Boolean value is tainted and assigned to a local variable *switch* inside the *onSensorChanged* method. Since *switch*'s value is control-dependent on *accelerationSquareRoot*, which is data-dependent on *event*, *switch* is tainted with the same mark. We specially taint the return value of a method if any of its arguments is tainted (including the implicit "this" argument if any), and we choose not to track other finer-grained control flows. This is because tracking control flows can cause significant performance overhead and imprecision in tainting [7]. Our taint propagation terminates at the call boundary of the corresponding sensor event handlers (e.g., the *onSensorChanged* handler). By this, we trace sensory data usage when an application executes.

3) *Analyzing Sensory Data Utilization*

From its initial state, an Android application visits a set of states by handling received events, and terminates at the final state when all its components are destroyed. The application state space can be unbounded. As explained earlier, we execute an application a finite number of times, and generate a distinct sequence of user interaction events for each execution. From this, we systematically explore different application states. We denote such explored state space as *S*. For each state, we analyze how sensory data are utilized, and compare their usage across different states in *S*. We define our metric of *utilization coefficient* by Equation (1):

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

The utilization coefficient of sensory data *d* at state *s* is defined as the ratio between data *d*'s usage at state *s* and the maximum usage of any sensory data in data pool *D* at any state in *S*. By this definition, a low utilization coefficient value indicates a low utilization of sensory data. The usage of sensory data *d* at state *s* is further defined by Equation (2):

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

Instr(*s, d*) is the set of bytecode instructions that are executed after the tainted sensory data *d* are fed to the application at state *s* until the taint propagation terminates. Boolean function *rel*(*i*) tests whether an instruction *i* uses any program data tainted by the same mark (1 for yes and 0 for no). Function *weight*(*i,s*) assigns a weight to instruction *i* based on its type and the state at which *i* is executed. We follow the principle that an instruction should have a higher weight if its

Table 3. GPS data utilization coefficients at three states

Application State ("→" means "followed by")	GPS Data Utilization Coefficient
$HS_0 \rightarrow HS_1$	$6n/6n = 1.00$
$HS_0 \rightarrow HS_2$	$3n/6n = 0.50$
$HS_0 \rightarrow HS_1 \rightarrow HS_2$	$4n/6n = 0.67$

❑ HS_0 : the handler sequence that initializes the map activity
❑ HS_1 : the handler sequence that handles the event "enable location tracking"
❑ HS_2 : the handler sequence that handles the event "switch activity"

execution brings more benefits to users, and formulated the following heuristic weight assignment strategy:

- Any unary, binary computation, or assignment instruction has a unit weight, no matter at which state the instruction is executed.
- The weight of a method-calling instruction *j* depends on the number of instructions (say, *n*) in that method and the state when *j* is executed. If the method is to update invisible GUI elements at background, *j*'s weight is set to *–n*. Otherwise, it is set to *n*.
- Any other instruction has a zero weight.

Based on sensory data utilization coefficients, one can identify those application states where sensory data are underutilized. Let us consider three states of our motivating example. They are listed in Table 3 and represented by the sequences of event handlers called after the application launches. For example, the third state ($HS_0 \rightarrow HS_1 \rightarrow HS_2$) means that users enable location tracking (HS_1) after the application initializes its *MapActivity* (HS_0); after a while, they switch to other activities (HS_2). Let us analyze sensory data utilization for the three states. For ease of presentation, we explain at the source code level and assume each method contains *n* instructions. Consider the second state where users switch to other activities after using *MapActivity*, and the location tracking is disabled by default. At this state, GPS data and their dependent data are consumed by *processLocation*, *putExtra*, *sendBroadcast*, *getExtra*, and *updateMap* method calls in turn. Boolean function *rel* returns 1 for all these method call instructions, and their weights are all *n* except *updateMap* whose weight is *–n* since this method is used to render an invisible map. According to Equation (2), GPS data usage at this state is 3*n*. We can also calculate that GPS data have a maximum usage of 6*n* at the first state where all method call instructions have a weight of *n* (including *persistToDatabase*). Then, the GPS data utilization coefficient at the second state is 0.50 (3*n*/6*n*). GPS data utilization coefficients at other states can be calculated similarly. The results reveal that GPS data are the least utilized at the second state. Our approach allows such automated analysis and reports energy inefficiency findings.

Our tool implementation ranks sensory data utilization coefficients at different states so that energy inefficiency reports can be prioritized. These reports contain information about how sensory data are used at each state, and highlight those method call instructions with high or negative weights. Our tool also provides detailed event handler calling traces to help developers construct concrete test cases to reproduce specific sensory data utilization scenarios. This actionable information can effectively help locate energy inefficiency problems in Android applications.

IV. EVALUATION

We implemented our approach as a prototype tool called GreenDroid on top of JPF. Implementation details can be found in our technical report [24]. In this section, we evaluate our approach by controlled experiments. We aim to answer the following two research questions:

- **RQ1:** *Can our approach effectively detect energy inefficiency problems in Android applications?*
- **RQ2:** *How does our approach compare with existing resource leak detection techniques?*

A. Experimental Setup and Design

We selected six popular open-source Android applications as our experimental subjects. They are available on Google Play Store or Google Code. Table 4 lists their basic information. All applications were compiled for Android 2.3.3. We conducted our experiments on a dual-core machine with Intel Core i5 CPU and 8GB RAM, running Windows 7.

To answer research question RQ1, we ran GreenDroid to analyze each application for its sensory data utilization at different states. We controlled GreenDroid to generate at most six user interaction events during each application execution. This suffices for GreenDroid to explore considerable application states. We examined the analysis reports to see whether they helped locate real energy inefficiency problems in these applications. We report the results in Section IV.B.

Our work shares some similarities with existing resource leak detection techniques [3][25]. To answer research question RQ2, we compared our approach with them. Since existing techniques do not apply to Android applications, we reimplemented them on top of JPF. We conducted experiments with two settings. First, we applied these techniques without our AEM model, assuming that event handlers could be arbitrarily called. Second, we applied these techniques and enforced the event handler orderings required by our AEM model. Under both settings, we checked whether existing techniques helped locate energy inefficiency problems in the six applications. We report the results in Section IV.C.

B. Detected Energy Inefficiency Problems

Table 4 presents our tool's analysis overhead for the six applications. Even for the two largest subjects Omnidroid and DroidAR (over 18K LOC), our analysis finished within five minutes and cost less than 600 MB memory. Such overhead is well supported by modern PCs and compares favorably with state-of-the-art testing or debugging techniques, which typically take hours to explore up to 100K states [23]. Our tool found real energy inefficiency problems in the six applications[3]. The problems in Osmdroid, Zmanim, DroidAR and Recycle-locator have been confirmed by developers prior to our experiments. The problems in Omnidroid and GPSLogger are new, and one of them has been recently confirmed by developers. We present these findings below.

Osmdroid. Osmdroid is an application similar to Google Maps. After analysis, our tool reported that its location data utilization coefficient is smaller than 0.4453 at 49.95% of all explored states as shown in Figure 5(a). This strongly sug-

[3] All our reported issues can be found on Google Code: http://code.google.com

Table 4. Application information and analysis overhead

Application	Basic Information				Analysis Overhead		
	Revision No.	Lines of code	Downloads	Availability	Explored states	Time (seconds)	Space (MB)
Osmdroid	750	18,091	10,000—50,000	Google Play Store	120,189	151	591
Zmanim	322	4,893	10,000—50,000	Google Play Store	54,270	110	205
Omnidroid	863	12,427	1000—5000	Google Play Store	52,805	220	342
DroidAR	204	18,106	1000—5000	Google Code	91,170	276	217
Recycle-locator	68	3,241	1000—5000	Google Play Store	114,709	43	153
GPSLogger	15	659	1000—5000	Google Code	58,824	35	149

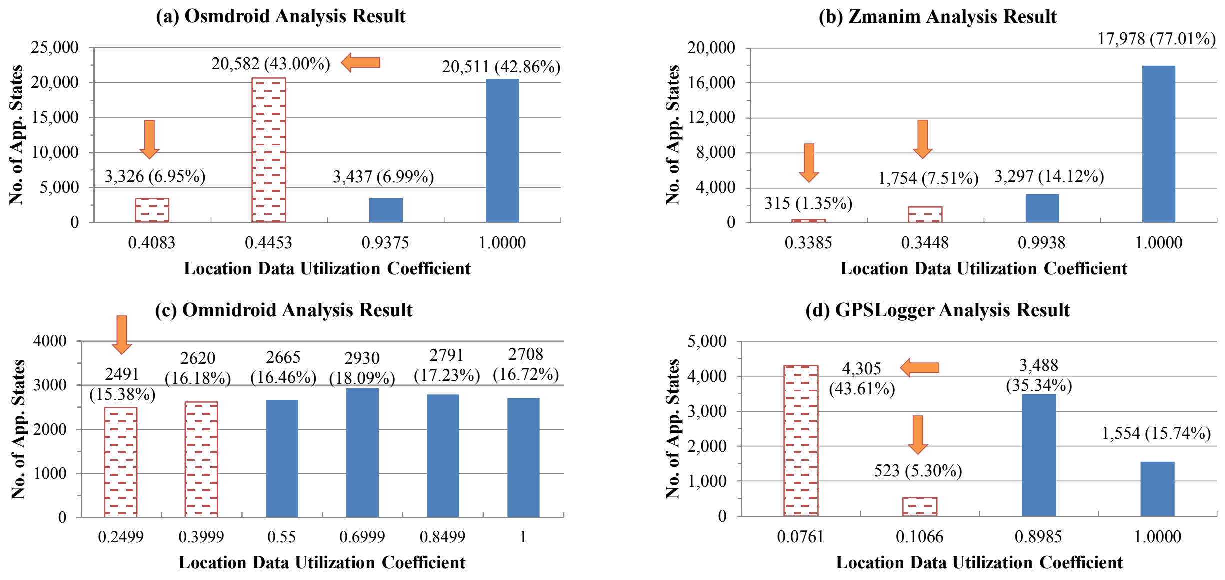

Figure 5. Utilization analysis results of location data

gests that Osmdroid has energy inefficiency problems. From the top ranked 15 reports, we can quickly find that if users switch from MapActivity to other activities without enabling location tracking, location data would be used to render an invisible map. This greatly wastes valuable battery energy as already reported by users (Osmdroid Issue 53).

Zmanim. Zmanim is a location-aware application for displaying prayer times during a day (zmanim) for Jews. The application generates zmanim according to users' location. Interestingly, developers realized that location sensing is energy-consuming, and they made the application stop location sensing once required location is obtained. However, as Figure 5(b) shows, our tool reported that at 8.86% of all explored states, the location data utilization coefficient is smaller than 0.3448. We confirmed that this energy inefficiency problem is similar to that found in Osmdroid. If users switch from the location sensing activity to other activities before a required location is obtained, battery energy would be wasted to update invisible GUI elements. In areas where GPS signals are weak, users repeatedly complained that Zmanim caused huge battery drain when it failed to detect required locations in a short time (Zmanim Issues 50 and 56).

Omnidroid. Omnidroid helps automate system functionalities based on user contexts. For example, Omnidroid can help users automatically reply a message like "busy in meeting" when the user receives a phone call during an important meeting. When Omnidroid runs, it maintains a background service to periodically check location updates. If any location update satisfies a pre-specified condition, its corresponding action would be executed. Our analysis result in Figure 5(c) shows that some states have a location data utilization coefficient of only 0.2499. We found that at these states, users actually have not yet specified any conditions. In other words, location data are collected for no use. Then why does the background service keep collecting location data? It is a huge energy waste. We reported our finding to Omnidroid developers, and suggested enabling location sensing only when there are conditions concerning user locations. We received an enthusiastic confirmation (Omnidroid Issue 179):

"Completely true, and your suggestion is a great idea and you're correct omnidroid does suck up way more energy than necessary as a result. I'd be happy to accept a patch in this regard".

GPSLogger. GPSLogger collects users' GPS coordinates to help them tag photos or view their tracks in Google Earth. Figure 5(d) presents its GPS data utilization analysis result. We found that at 43.61% of all explored states, GPS data are extremely underutilized (the utilization coefficient is only 0.0761). GPSLogger maintains a background service to retrieve GPS data and evaluate whether they satisfy certain

Table 5. Random execution result

App Name	Exceptions	Feasible Executions	App Name	Exceptions	Feasible Executions
Osmdroid	79/100	0/100	DroidAR	67/100	0/100
Zmanim	31/100	0/100	Recycle-locator	4/100	3/100
Omnidroid	22/100	0/100	GPSLogger	9/100	1/100

precision requirements. If yes, they are processed and stored in a database. Otherwise, they are discarded. When GPS signals are weak, GPS sensors may keep generating noisy data. Such produced GPS data are mostly discarded and cause energy waste, which is why GPS data have a very low utilization at some application states. For this problem, we suggested temporarily disabling location sensing if the application finds that GPS data keep being of low quality (GPSLogger Issue 7). This reported issue is pending.

Our tool also located energy wastes in DroidAR and Recycle-locator. DroidAR is a framework for augmented reality on Android. It leverages sensory data to digitalize the real world and make users' environment interactive. Recycle-locator is a location-aware application for helping users quickly locate services in university campuses. Our tool detected that their location listeners are never unregistered after usage (DroidAR Issue 27 and Recycle-locator Issue 33). This poses big threats to a battery's lifetime as these sensors may keep running until the application process is killed [2].

C. Comparision with Existing Techniques

Without our AEM model, JPF would encounter great challenges in analyzing Android applications. Table 5 lists such results. JPF executed each application 100 times and event handlers were called randomly, i.e., not following the orderings required by our AEM model. We observe many runtime exceptions in these executions. For example, 79 out of 100 executions of Osmdroid failed because of such exceptions. This arises from the ignorance of the data flow dependencies between event handlers. For our two small-sized subjects Recycle-locator and GPSLogger, JPF threw fewer exceptions because the data flows between event handlers are relatively simple. Still, few (1–3) of their executions can occur in reality (feasible), and this seriously hinders the application of existing resource leak detection techniques.

For comparison, we enabled our AEM model for JPF, and applied existing resource leak detection techniques to analyze these applications. All executions then became feasible, and the energy inefficiency problems related to sensor listener misusage in DroidAR and Recycle-locator were located. Still, we note that these techniques could not locate other energy inefficiency problems related to sensory data underutilization. Achieving that requires non-trivial data flow tracking and analysis for sensory data, which are not available in existing resource leak detection techniques.

D. Discussion

Our approach is independent of its underlying program analysis framework. Currently, we implemented it on top of JPF because there is no other suitable tool but JPF is a highly extensive Java program verification framework (Android programs are written in Java). Still, analyzing Android applications using JPF is challenging. First, we need to derive an extensible AEM model from Android specifications and enforce it to make JPF call event handlers in a reasonable way. Second, some Android APIs rely on native libraries whose implementation is specific to hardware and operating systems. The semantics of these APIs and their enclosing library classes have to be properly modeled due to JPF's closed-world assumption. Modeling Android libraries is known to be a difficult and tedious task [14]. Our current implementation only considered a critical subset of library classes and concerned APIs and can already analyze many real-world Android applications [8]. Extending our tool to support more Android APIs is possible and we are exactly on this way.

Our work has some limitations. First, our approach currently cannot simulate complex user inputs such as texts (e.g., password) nor systematically generate mock sensory data. We will study the effects of this limitation in future. Besides, we are not clear whether our approach can be easily generalized to help diagnose other types, especially unknown types of energy waste. We also make this our future work.

V. Related Work

Our work relates to energy efficiency analysis, resource leak detection, and information flow tracking. We discuss some representative work below.

Energy efficiency analysis. In the past several years, researchers proposed various techniques to help improve smartphone applications' energy efficiency. Kim et al. proposed to use power signatures based on system hardware states to detect energy-greedy malwares [27]. Pathak et al. characterized energy bugs in smartphone applications [22]. They also proposed eProf to help estimate an application's energy consumption by tracking the activities of energy-consuming entities when an application runs on mobile devices [21]. WattsOn [13] shares a similar spirit with eProf but enables energy emulation on the developers' workstations. MAUI [5] helped offload "energy-consuming" tasks to resource-rich infrastructures. EnTracked [12] and RAPS [18] adopted different heuristics to guide an application to use GPS sensors in a smart way. Little Rock [19] suggested a low-power processor for energy-consuming sensing operations. SALSA [20] helped select optimal data links for saving energy in large data transmission. Our work shares a similar goal, but focuses on detecting energy inefficiency problems by systematically exploring different application states and diagnosing sensory data utilization. Our work incurs reasonable overhead, and provides developers with actionable information to locate energy inefficiency problems.

Resource leak detection. System resources are finite, and developers have to ensure acquired resources to be released eventually. This task is error-prone. Empirical evidence shows that resource leaks commonly occur [29]. Researchers proposed language-level mechanisms and automated management techniques to prevent such leaks [6]. Various tools were also designed to detect resource leaks [3][25]. For example, QVM [3] is a specialized runtime environment for detecting defects in Java programs. It monitors application executions and checks for violations of resource safety policies. TRACKER [25] is an industrial-strength tool for finding resource leaks in Java programs. It conducts inter-procedural static analysis to ensure no resource safety policy is violated on any execution path. Our experiments have shown that these resource leak detection techniques can only help reveal trivial energy inefficiency problems.

Information flow tracking. Dynamic information flow tracking (DFT) tracks interesting data as they propagate in program execution [11]. DFT has many applications. TaintCheck [10] uses DFT to protect commodity software from memory corruption attacks such as buffer overflows. It taints input data from untrusted sources and ensures that they are not used in a dangerous way. TaintDroid [7] prevents Android applications from leaking users' private data. It tracks the data from privacy-sensitive sources and would warn users when such data leave the system. LEAKPOINT [4] leverages DFT to help fix memory leak in C and C++ programs. Dynamically allocated memory blocks are tainted and monitored in case that they are forgotten to release.

VI. CONCLUDING REMARKS

In this paper, we have presented an approach for diagnosing energy inefficiency problems in Android applications. Our approach simulates the runtime behavior of an application, and automatically analyzes its sensory data utilization. It helps developers locate energy inefficiency problems caused by misusage of sensors and their data underutilization. We implemented a tool on top of JPF, and evaluated it using six real Android applications. The results confirmed its effectiveness in locating energy inefficiency problems.

In our on-going research, we are extending this work in two directions. First, we are investigating more sensor-based Android applications to study whether misusage of sensors and their data underutilization are two dominant causes of energy waste. Second, we are investigating other energy inefficiency problems not related to sensors. For example, our initial survey and related work [26] both suggest that Android's wake lock mechanism can also cause severe energy waste. We believe that our work together with other related ones will make smartphone applications more energy efficient, and this can benefit millions of users.

ACKNOWLEDGEMENT

This research was partially funded by Research Grants Council (General Research Fund 612210, 612309) of Hong Kong, and by National Basic Research Program (973 program 2009CB320702), National High-Tech Research & Development Program (863 program 2011AA010103), and National Natural Science Foundation (61100038, 61021062) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

REFERENCES

[1] "Android Sensor Management." URL: http://developer.android.com/reference/android/hardware/SensorManager.html

[2] "Android Process Lifecycle." URL: http://developer.android.com/reference/android/app/Activity.html#ProcessLifecycle.

[3] M. Arnold, M. Vechev, and E. Yahav, "QVM: an efficient runtime for detecting defects in deployed systems," ACM Trans. Sotware Engineering and Methodology, vol. 21, 2011, pp. 2:1-2:35.

[4] J. Clause and A. Orso, "LEAKPOINT: pinpointing the causes of memory leaks," Proc. Int'l Conf. Soft. Engr. (ICSE 10), pp. 515-524.

[5] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 10), ACM, 2010, pp. 49-62.

[6] I. Dillig, T. Dillig, E. Yahav, and S. Chandra, "The CLOSER: automating resource management in Java," Proc. Int'l Symp. Memory Management (ISMM 08), ACM, 2008, pp. 1-10.

[7] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," Proc. USENIX Conf. Operating Systems Design and Impl. (OSDI 10), 2010, pp. 393-407.

[8] "GreenDroid Project". Available: http://www.cse.ust.hk/~andrewust/

[9] "JPF and Google Summer of Code." URL: http://babelfish.arc.nasa.gov/trac/jpf/wiki/events/soc2012

[10] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Proc. ISOC Network and Distributed System Security Symp., 2005.

[11] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," Proc. ACM Conf. Virtual Exe. Env. (VEE 12), 2012, pp. 121-132.

[12] M. B. Kjærgaard, J. Langdal, T. Godsk, and T. Toftkjær, "EnTracked: energy-efficient robust position tracking for mobile devices," Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 09), ACM, 2009, pp. 221-234.

[13] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," Proc. 18th Int'l Conf. Mobile Computing and Networking (Mobicom 12), 2012, pp. 317-328.

[14] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," SIGSOFT Softw. Eng. Notes, vol. 37, 2012, pp. 1-5.

[15] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," Proc. ACM SIGSOFT Int'l Symp. Foundations of Soft. Engr. (FSE 12), ACM, 2012.

[16] "Osmand Issue 1092." Available: http://code.google.com

[17] "Osmdroid Issue 53." Available: http://code.google.com/

[18] J. Paek, J. Kim, and R. Govindan, "Energy-efficient rate-adaptive GPS-based positioning for smartphones," Proc. Int'l Conf. Mobile Systems, App., and Services (MobiSys 10), ACM, 2010, pp. 299-314.

[19] B. Priyantha, D. Lymberopoulos, and J. Liu, "LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones," IEEE Pervasive Computing, vol. 10, 2011, pp. 12-15.

[20] M. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely, "Energy-delay tradeoffs in smartphone applications," Proc. Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 10), ACM, 2010, pp. 255-270.

[21] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof," Proc. Euro. Conf. Comp. Sys. (EuroSys 12). 2012, pp. 29-42.

[22] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps," Proc. 10th Int'l Conf. Mobile Systems, Applications, and Services (MobiSys 12), 2012, pp. 267-280.

[23] "Robotium, a testing framework for Android applications." URL: http://code.google.com/p/robotium/.

[24] Y. Liu, C. Xu and S.C. Cheung, "Verifying Android applications Using Java Pathfinder," Technical Report HKUST-CS-12-03.

[25] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," Proc. Int'l Conf. Soft. Engr. (ICSE 10), 2010, pp. 535-544.

[26] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying android apps for the absence of no-sleep energy bugs," Proc. USENIX Conf. Power-Aware Comp. and Sys. (HotPower 12), 2012.

[27] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," Proc. Int'l Conf. Mobile Sys., App's, and Services (MobiSys 08), 2008, pp. 239-252.

[28] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," Proc. Int'l Conf. Automated Soft. Engr., 2000, pp. 3-11.

[29] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," Proc. ACM SIGPLAN Conf. Object-oriented Prog., Sys, Lang., and App's (OOPSLA 04), 2004, pp. 419-431.