SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2017, Vol. 60 012104:1–012104:18
doi: 10.1007/s11432-015-1015-5

CyanDroid: stable and effective energy inefficiency diagnosis for Android apps

Qiwei LI[1,2], Chang XU[1,2]*, Yepang LIU[3], Chun CAO[1,2],
Xiaoxing MA[1,2] & Jian LÜ[1,2]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023*, China;*
[2]*Department of Computer Science and Technology, Nanjing University, Nanjing* 210023*, China;*
[3]*Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China*

Received May 17, 2016; accepted August 12, 2016; published online November 17, 2016

Abstract Smartphones are an indispensable part of people's daily lives. Smartphone apps often use phone sensors to probe their users' physical environmental conditions to provide services. However, sensing operations can be energy-consumptive, and thus the obtained sensory data should be effectively utilized by apps for their users' benefits. Existing studies disclosed that many real-world smartphone apps have poor utilization of sensory data, and this causes serious energy waste. To diagnose such energy bugs, a recent technique GreenDroid automatically generates sensory data, tracks their propagation and analyzes their utilization in an app. However, we observe that GreenDroid's sensory data generation is random and this can negatively affect its stability and effectiveness. Our study reported that GreenDroid might miss energy bugs that require specific sensory data to manifest. To address this problem, we propose a novel approach to systematically generating multi-dimensional sensory data. For effective diagnosis, we also propose to consider app state changes at a finer granularity. We implemented our approach as a prototype tool CyanDroid, and evaluated it using four real-world Android apps and hundreds of their mutants. Our results confirmed that CyanDroid is more stable and effective in energy inefficiency diagnosis for sensory data underutilization issues.

Keywords smartphone, sensory data utilization, energy inefficiency, white-box sampling, dynamic tainting

Citation Li Q W, Xu C, Liu Y P, et al. CyanDroid: stable and effective energy inefficiency diagnosis for Android apps. Sci China Inf Sci, 2017, 60(1): 012104, doi: 10.1007/s11432-015-1015-5

1 Introduction

With the rapid development of mobile technologies, the number of smartphone users grows exponentially in recent years. By the end of year 2014, there had been more than 1.6 billion smartphones shipped around the world[1]. These smartphones are equipped with various sensors, such as GPS and accelerometer, so that developers can exploit them to build apps that are able to provide context-aware services based on users' physical and cyber environmental conditions. For example, the app Nike+ Running can record

* Corresponding author (email: changxu@nju.edu.cn)
1) Number of smartphone users worldwide. http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694.

its users' running routes and speeds by location sensing for helping them do exercises in a scientific and healthy way2).

It is well-known that sensing operations can be energy-consumptive [1], but smartphone battery power is very limited, instead. This requires smartphone apps to have to use sensors in an energy-efficient way. Otherwise, phone batteries can be quickly depleted, causing bad user experiences. Unfortunately, many apps on market fail to use sensors energy-efficiently [2, 3]. They frequently conduct sensing operations, which incur high energy cost, but do not effectively use the obtained sensory data to bring users observable benefits. Such problems are known as "sensory data underutilization energy bugs" [3]. For example, many poorly-implemented apps only use sensory data to update invisible GUI elements when they are switched to background. Such usage of sensors clearly wastes precious battery power and is thus undesired. Developers are suggested to deactivate sensors in such problematic scenarios.

To help developers diagnose energy bugs caused by sensory data underutilization, Liu et al. proposed an automated technique named GreenDroid [3]. GreenDroid tracks the transformation, propagation and consumption of sensory data, and analyzes whether the data are cost-effectively utilized by an Android app at different app states. However, we observed two limitations from GreenDroid. First, it generates mock sensory data in a random manner and feeds them to a running app for sensory data utilization analysis. Such randomness can prevent GreenDroid from reaching certain app states (e.g., those that are only reachable by specific sensory data inputs) during analysis (see later Section 2 for an example). Second, GreenDroid considers that an app transfers to a new state after it finishes handling an event (e.g., a button click), regardless of which program path is executed during this event handling. However, different execution paths can lead to different app states, which are not distinguished or controlled in GreenDroid. These limitations can negatively affect the stability and effectiveness of GreenDroid's energy inefficiency diagnosis.

To facilitate effective energy inefficiency diagnosis, we in this article extend GreenDroid, aiming to address the aforementioned limitations. Regarding the first limitation, we propose a novel and efficient approach to systematically generating sensory data. Our sensory data generation is based on the white-box sampling [4], which was proposed to address the uncertainty of input data for traditional programs. This technique cannot be directly applied to sampling sensory data since it focuses on program inputs with a single dimension, while sensory data are typically multidimensional (e.g., a location datum contains latitude, longitude, altitude three parts, and an app can even use several types of sensory data). Therefore, we adapt the traditional white-box sampling to support program data with arbitrary dimensions. As such, our approach can systematically generate sensory data to reach different app states and search problematic ones. Regarding the second limitation, we refine the concept of app state in GreenDroid to a finer-grained level. Specifically, when handling multiple sensory events (e.g., sensory data update events), if an app executes different paths of a handler, we would consider that the app goes to different states rather than the same one, which is GreenDroid's treatment, after it finishes the event handling.

By addressing the two GreenDroid's limitations, we aim to enhance its stability and effectiveness in energy inefficiency diagnosis for smartphone apps. In summary, this article makes the following contributions:

(1) We adapted the traditional white-box sampling technique to handle multidimensional program data and applied it to analyze sensory data utilization for smartphone apps.

(2) We refined the app state concept in smartphone app analysis, and this refinement is necessary for improving the stability and effectiveness to an approach's energy inefficiency diagnosis.

(3) We implemented our approach as a prototype tool named CyanDroid for the Android platform. We evaluated it with four popular Android apps and their 522 energy issue related mutants. Our results confirmed CyanDroid's stability, effectiveness and efficiency in energy inefficiency diagnosis. We also conducted a case study and found two critical energy bugs in the original apps.

The rest of this article is organized as follows. Section 2 gives a motivating example. Sections 3 and 4 present our energy inefficiency diagnosis approach. Section 5 evaluates our approach and discusses its

2) Nike+ Running. http://www.nike.com/us/en_us/c/running/nikeplus/gps-app.

```
public void onLocationChanged(Location loc) {
   double dis = calDistance(loc, goalLoc);
   if (dis < NEAR_DISTANCE)
      doLightWork();
   else if (dis < NOT_FAR_DISTANCE)
      doHeavyWork();
   else
      ; //do nothing
}
```

Figure 1 (Color online) Simplified motivating example.

experimental results. Section 6 reviews related work, and finally Section 7 concludes this article.

2 Motivation

Figure 1 gives a simplified motivating example of energy inefficiency caused by sensory data underutilization. The concerned app listens to GPS updates and defines a handler `onLocationChanged` to handle location changes. Every time when there is a location change, the `onLocationChanged` handler will calculate the distance between the new location and a target location. Based on this calculated distance, the handler then conducts different tasks. If the distance is less than `NEAR_DISTANCE`, which means that the new location is close to the target location, the handler will conduct some light-weight processing of the data (processing detail is simplified). If the distance is larger than `NEAR_DISTANCE` but less than `NOT_FAR_DISTANCE`, which means that the new location is not far away from the target location, the handler will conduct heavy-weight processing instead. Finally, if the distance is larger than `NOT_FAR_DISTANCE`, which means that the new location is far away from the target location, the handler will do nothing. This simplified example is adapted from real apps and represents their common sensory data processing flows (e.g., GeoPointer[3)]).

From this example, we observe that whether a specific location is close to the target location will decide how the concerned location datum is processed, and this then affects whether the datum can be well utilized and bring desirable benefits to users. For this example, all location data that indicate locations far away from the target location will always be discarded and not bring any perceptible benefit. If an app frequently encounters such cases and keeps obtaining and discarding location data (e.g., when its user cannot get close to the target location in short time), the location data utilization would be very low compared to location data that are close to the target location, causing significant energy waste. The users can regard this as an energy inefficiency bug. To address this inefficiency, developers should reduce the frequency to acquire location data when its user is far way from the target location.

To diagnose such energy inefficiency bugs, developers have to carefully design and systematically generate various location data to cover all data usage scenarios. This can be non-trivial for human developers. To address this challenge, Liu et al. proposed an automated testing technique, GreenDroid, to assist developers to diagnose sensory data underutilization problems [4,5]. The technique generates mock sensory data, feeds them into a running app, and tracks the propagation of the data to analyze whether they are effectively utilized. However, we observe that GreenDroid's sensory data generation is essentially random, and due to this randomness, it might only be able to cover few sensory data usage scenarios when analyzing apps such as our motivating example. This would lead to unstable analysis results (e.g., when randomly executing different program paths in different runs) and cause GreenDroid to miss potential energy inefficiency bugs, reducing its energy diagnosis effectiveness.

Therefore, this example motivates us to propose a new approach to addressing GreenDroid's limitation in terms of replacing random sensory data generation by guided sensory data generation. Actually, our approach generates sensory data with multidimensional white-box sampling to cover as many scenarios

3) GeoPointer. https://code.google.com/p/geo-pointer/.

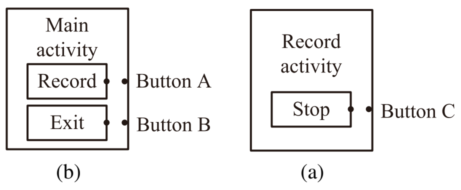

Figure 2 Example app. (a) Main activity; (b) record activity.

as possible. Our analysis in Subsection 3.3 shows that it can quickly cover scenarios in the motivating example using very few sensory data points.

3 Methodology

We in this section elaborate on our energy inefficiency diagnosis approach. We first introduce how we refine the state concept in GreenDroid, and then discuss how we systematically generate different sensory data to cover different execution paths in an Android app.

3.1 State space

While our energy inefficiency diagnosis approach generally applies to smartphone apps, we in this article restrict our discussions to Android apps for specific treatment of code and GUI styles. Android apps are event-driven. An app's program logic is mainly defined by a set of loosely-coupled handlers. At runtime, they are invoked by the Android OS to handle various events. For example, if a user clicks a button, the Android OS will invoke the button's corresponding handler to handle this button click event. For testing/analysis purposes, an Android app's behavior can be modelled as a state machine [5]. When an event occurs, an app's state will transit to a new one after the app handles the event (since this also changes the app's internal data). In order to detect bugs, which often only manifest at certain application states, automated testing/analysis techniques would have to generate a large number of event sequences to exercise an Android app to explore its different states.

Similar to GreenDroid, our approach analyzes an Android app to learn its registered handlers and generates event sequences to exercise the app for energy inefficiency diagnosis. One major difference between our approach and GreenDroid is the granularity of application states analyzed. GreenDroid considers that an app transits to a new state after invoking a certain handler to handle a sensory event. This state is associated with this handler, no matter what sensory data the event contains and how the event is handled. This is fine for efficiency consideration, and some energy inefficiency bugs that do not rely on specific sensory data can indeed be well detected. However, when fed with different sensory data, an event handler may follow different execution paths, thus changing the app's state in different ways. If one does not distinguish such different cases, it may potentially miss those energy inefficiency bugs that can manifest only with specific sensory data. Therefore, we need to refine the state concept in GreenDroid to a finer-grained level to overcome this limitation. Specifically, if an app executes on different paths in a handler, our approach would consider it going to different new states.

Figure 2 gives a simple example to illustrate the app state concept in our approach and its difference from that in GreenDroid. The example app contains two different screen GUIs: `Main activity` and `Record activity`. After launch, the app's initial GUI is set to the `Main activity`, which contains two buttons: `Record` and `Exit`. If its user clicks the `Exit` button, the app terminates immediately. If the user clicks the `Record` button, the app registers a GPS event listener to handle location changes and switches to the `Record activity`. When handling location changes, the app checks whether new location data satisfy certain precision requirements. If yes, the app stores them to a database. Otherwise, the data are discarded. There is a `Stop` button on the `Record activity`'s GUI for stopping listening to GPS updates and returning to the `Main activity`. Figure 3(a) gives the app's state transition graph in GreenDroid, and Figure 3(b) gives the counterpart in our approach. The difference lies in state S_1, where the `Record activity` handles location data. In Figure 3(a), the app transits from state S_1 to S_4 independent of what

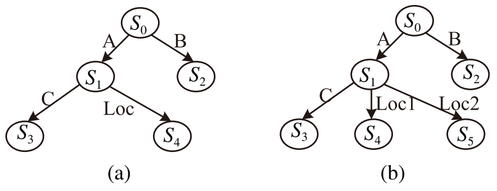

Figure 3 State space illustration. (a) Traditional state space; (b) our state space.

location data are handled. Nevertheless, in our approach the app transits from S_1 to S_4 only if it handles precise location data (storing them to a database), and to S_5 when the app encounters imprecise location data (discarding them). We observe that if a user stays in an environment with bad GPS signal, the app would often reach state S_5. We understand that continuously discarding sensed GPS data can be a great waste of batter power. As such, an effective energy inefficiency diagnosis technique should be able to distinguish state S_5 from S_4 and identify it in a systematic way. Unfortunately, GreenDroid cannot do so since its explored state S_4 might not be problematic if its randomly generated sensory data happen to be precise. This explains why we need to refine its state concept for effective energy inefficiency diagnosis.

3.2 White-box sampling

To diagnose energy inefficiency effectively, one needs to generate various sensory data to cover different data usage scenarios in a systematic way. One way is to apply dynamic symbolic execution [6]. However, it may not be practical since real-world Android apps can contain many non-linear path constraints that cannot be effectively solved by existing SMT solvers. Therefore, we aim to address this issue by adapting the white-box sampling idea [4], which was originally proposed to address the uncertainty of input data for conventional programs.

White-box sampling samples continuous data ranges and selects representative data points as inputs to test a program. Its goal is to cover different execution paths of a program and approximate its functionality at a low cost. The technique considers critical instructions of a program as hash indexes, such as branching and type conversion instructions, because their execution results largely decide which path to proceed next or which functionality to select at runtime. Specifically, given an input x, its hash index h_x is a series of critical instructions that are visited in this input's induced program execution. The technique follows Algorithm 1 to select new data points from past executions. We explain its idea below.

Algorithm 1 White-box sampling

```
Input: (min, max): data range;
 1: stack ← $\phi$;
 2: $h_{\min}$ = Execute(min);
 3: $h_{\max}$ = Execute(max);
 4: stack.push(min, $h_{\min}$, max, $h_{\max}$);
 5: while !stack.isEmpty() do
 6:     $(a, h_a, b, h_b)$ = stack.pop();
 7:     if ($h_a == h_b$ and $|b - a| >$ Threshold_L)
        or ($h_a \neq h_b$ and $|b - a| >$ Threshold_S) then
 8:         $h_m$ = Execute($(a + b)/2$);
 9:         stack.push($a, h_a, (a + b)/2, h_m$);
10:         stack.push($(a + b)/2, h_m, b, h_b$);
11:     end if
12: end while
```

The algorithm first feeds min and max to the program under test to obtain their hash indexes $h_{\min}$ and $h_{\max}$ by calling function `Execute` (Lines 2–3). Then it pushes $(\min, h_{\min}, \max, h_{\max})$ into stack as the first pair of sampled data points (Line 4), and enters a loop (Lines 5–12). At each iteration of the loop, it pops out a record (a, h_a, b, h_b) from stack (Line 6). If the two hash indexes h_a and h_b are identical but $|b-a|$ is larger than a threshold `Threshold_L` (meaning that although they seem to behave the same, their range can still be further split), or h_a and h_b are not identical and $|b - a|$ is larger than another threshold `Threshold_S` (meaning that they behave differently and their range is still large enough for further splitting), the algorithm would split the data range from a to b. This is done by calculating the middle data point m between a and b, and obtaining its corresponding hash index h_m. After that, the generated two pairs of data points, (a, h_a, m, h_m) and (m, h_m, b, h_b), are pushed into stack as new sampled data points (Lines 7–10). The loop will continue until stack becomes empty (Line 5). `Threshold_L` is used to ensure that one can sample enough data points for a large range, and `Threshold_S` is used to avoid sampling data points in a too small range.

White-box sampling requires the data being sampled to be continuous. Sensory data for Android apps happen to follow this requirement. However, applying white-box sampling to our problem still has challenges. Traditional white-box sampling can only handle program data with one dimension, but sensory data can be multidimensional. Actually, traditional white-box sampling technique supports multidimensional data by mapping multiple parameters to a single one with a hash function. It then applies its one-dimensional white-box sampling for later processing. The effectiveness of doing so would depend on how the hash function is designed. Therefore, we adapted the traditional white-box sampling to support multidimensional data directly instead of using a hash function. We explain in the following how to address it for systematic generation of sensory data.

3.3 Multidimensional white-box sampling

For an Android app, its processed sensory data often have multiple dimensions (e.g., GPS data have three dimensions: latitude, longitude and altitude). Therefore, we consider multidimensional white-box sampling for handling them.

3.3.1 *Input space of sensory data*

We first define a structure, `Space`, to represent the value space of sensory data. `Space` consists of multiple variables, labeled with their value ranges. For example, `Space` $\{(\text{lat: } -180, 180), (\text{lon: } -90, 90)\}$ consists of two variables, lat and lon, for sampling of their possible values (lat can take any value from $[-180, 180]$, and lon can from $[-90, 90]$).

We introduce a new function, `ExecuteAll`, for feeding sampled data points to an Android app for execution. These sampled data points are vertexes of a `Space` and represent minimal or maximal values of respective variables in this `Space`. Each n-dimensional `Space` has 2^n such vertexes, i.e., $\{(a_1/b_1), \ldots, (a_n/b_n)\}$, where a_i/b_i represents the minimal/maximal value for the i-th dimension. For example, the earlier `Space` $\{(\text{lat: } -180, 180), (\text{lon: } -90, 90)\}$ consists of four vertexes $\{(-180, -90), (-180, 90), (180, -90), (180, 90)\}$. Function `ExecuteAll` feeds such a set of vertexes to an Android app as its input sensory data in a way the app accepts. When an event handler processes the sensory data, `ExecuteAll` collects their corresponding hash indexes from executions, similar to what traditional white-box sampling does.

We can split a `Space` into two sub-spaces with respect to a variable x in this space. Generally, for a `Space` $\{(x_i\text{: } a_i, b_i), i \in [1, n]\}$ with n variables, if we split it according to the variable x_j, whose value can range from a_j to b_j, the resulting two sub-spaces are `Space` $\{(x_i\text{: } a_i, b_i), i \in [1, j) \vee i \in (j, n] \cup (x_j\text{: } a_j, m_j)\}$ and $\{(x_i\text{: } a_i, b_i), i \in [1, j) \vee i \in (j, n] \cup (x_j\text{: } m_j, b_j)\}$, where m_j is $(a_j + b_j)/2$. When we split a n-dimensional `Space`, we get 2^{n-1} new vertexes. For our previous example, if we select variable lat to split `Space`, we obtain two sub-spaces $\{(\text{lat: } -180, 0), (\text{lon: } -90, 90)\}$ and $\{(\text{lat: } 0, 180), (\text{lon: } -90, 90)\}$, and the newly generated vertexes are $\{(0, 90), (0, -90)\}$.

We enable each `Space` with two methods named `cap` and `equal`. Method `cap` calculates the capacity of `Space`, i.e., $\prod_{i \in (1,n)} |b_i - a_i|$, which is the product of all variables' value ranges. The capacity represents the length for a one-dimensional `Space`, area for a two-dimensional `Space`, volume for a three-dimensional `Space`, and so on. Method `equal` checks whether the hash indexes of all vertexes in `Space` are identical. If yes, it returns `TRUE`, and otherwise, `FALSE`. We allow function `ExecuteAll` to run with a set of vertexes, and collect corresponding hash indexes and store them for later reference.

3.3.2 *Sampling algorithm*

Algorithm 2 describes how our multidimensional white-box sampling works. First, the algorithm executes an app with all vertexes of its app's initial space, which represents all possible sensory data, as the input sensory data (Lines 2–3). Then it pushes this initial space into stack (Lines 4) and enters a loop for space splitting (Lines 5–14). In each iteration of the loop, it pops out a `Space` and assigns it to a local variable now (Line 6). It then invokes now's two methods, `equal` and `cap`. If now.equal() returns `TRUE` (meaning that all vertexes' hash indexes are identical) but now's capacity is still larger than `Threshold_L` (meaning that the space is still very large), or now.equal() returns `FALSE` (meaning that all vertexes' has indexes are not identical) and now's capacity is still larger than `Threshold_S` (meaning that the space is large enough for further splitting), the algorithm continues to split the current space now (Line 7). It chooses variable d that has the maximal value range among all variables in `Space` now (Line 8), and splits now into two sub-spaces with respect to variable d (Line 9). After that, it generates new sample data newV from the two sub-spaces (Line 10), and feeds them to the app for execution as well as obtaining their corresponding hash indexes (Lines 11). Finally, the algorithm pushes the two sub-spaces into stack to continue its iterations (Line 12). Parameters `Threshold_L` and `Threshold_S` are the same as in traditional white-box sampling. If one expects more complete analysis, they can be set smaller.

Algorithm 2 Multidimensional white-box sampling

```
Input: space: data range
 1: stack = ∅;
 2: initV=space.allVertexes();
 3: ExecuteAll(initV);
 4: stack.push(space);
 5: while !stack.isEmpty() do
 6:     now=stack.pop();
 7:     if (now.equal() and now.cap()>Threshold_L) or (!now.equal() and now.cap()>Threshold_S) then
 8:         d=now.getVarWithMaxValueRange();
 9:         childSpaces=now.splitBy(d);
10:         newV=GetNewVertexes(childSpaces);
11:         ExecuteAll(newV);
12:         stack.pushAll(childSpaces);
13:     end if
14: end while
```

We apply the algorithm to our earlier motivating example. Let `Threshold_L` and `Threshold_S` be 1/4 and 1/32 of the initial input space's capacity, respectively. Then the multidimensional white-box sampling can generate 27 pieces of different location data. We show the sampled location data in Figure 4. The half-filled dot in circles represents the target location and the other dots represent the sampled data points. We observe that these sampled location data are representative: some are close to the target location in Figure 1 (i.e., those in the small circle, whose radius is `NEAR_DISTANCE`), some are not far away from the target location (i.e., those in the ring area between the small and large circles), and the others are far away from the target location (i.e., those outside the large circle, whose radius is `NOT_FAR_DISTANCE`). By feeding such location data to our motivating example, our energy inefficiency diagnosis can effectively cover and analyze all three different location data usage scenarios. As a comparison, if one uses random sampling, the chance of covering the three scenarios with a small number of sampled points (say 27) is very low.

3.4 Execution model

In order to apply multidimensional white-box sampling to analyzing smartphone apps, we define starting and ending points to calculate hash indexes. As mentioned earlier, we model a smartphone app as a state machine. When an app at state S_1 receives a sensory event E_1 it is listening to, the app will invoke its corresponding handler to handle the event. As such, the app is always waiting for inputs and handling

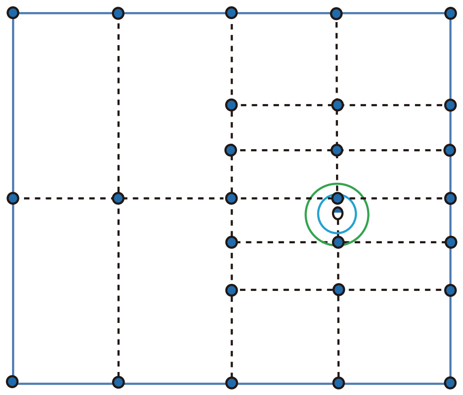

Figure 4 (Color online) Sampled data points for the motivating example.

them, and states like S_1 are similar to initial states of a traditional non-event driven program. Therefore, we consider entries of sensory event handlers in Android apps as starting points.

After an app finishes handling a sensory event, it transits to a new state. We consider the program points where an app finishes its handling of sensory events as ending points. There are two cases of ending points. First, if a sensory event handler does not create any new worker thread, the exit point of this handler is considered as its ending point. Second, if a handler creates a new worker thread to handle its received sensory event, the program point where the worker thread terminates is considered as its ending point. In practice, a worker thread may be long-running or even never terminate. Therefore, we can set a timeout limit for each sensory event handler. If it cannot finish handling its sensory event within the limit, we consider the program point where the timeout occurs as its ending point. With thus defined starting and ending points, one can calculate and record the hash index for each handler execution, which is a series of visited critical instructions, including branching and type conversion instructions. A detailed explanation of hash index can be found in related work [4].

4 Sensory data utilization analysis

As we discussed earlier, sensory data, which are obtained with high energy cost, should be well utilized by an app to bring users benefits. However, in many real-world Android apps, sensory data are ineffectively utilized or even discarded at certain app states, resulting in energy waste. In order to diagnose such energy inefficiency, we need to track how an app processes and utilizes its received sensory data at different app states. In this section, we elaborate on how to conduct the diagnosis. We made a slight modification of the method of GreedDroid and readers please refer to related work [2, 3] for a full treatment.

4.1 Tracking sensory data usage

After a sensory datum enters an app, it can be transformed into different forms and utilized by different app components. Therefore, the first step of energy inefficiency diagnosis is to track the transformation of the sensory datum, which can be done by the existing dynamic information flow tracking idea [7]. Specifically, after multidimensional white-box sampling generates a sensory datum, the datum object is associated with a unique taint mark and fed to the app under diagnosis. While the app transforms the sensory datum into other forms, the taint mark will be propagated to those new program data that are derived from the original sensory datum. Then when the app finishes handling its sensory datum, we can pinpoint all such tainted program data and record how they are used by the app (e.g., passed as an argument to an API call).

4.2 Analyzing sensory data utilization

When exploring an app's state space, we should record how sensory data are utilized for a sensor event.

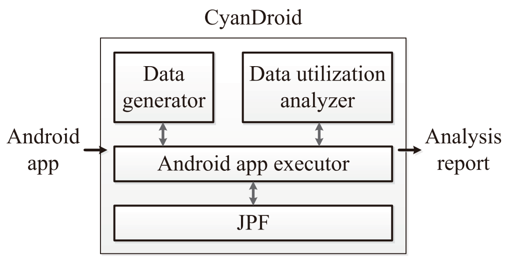

Figure 5 System architecture of CyanDroid.

For the state transition from S_1 to S_2 due to handling sensory data, we calculate a metric, relative sensory `Data Utilization Coefficient` (DUC), which is defined as follow:

$$\mathrm{duc}(T) = \frac{\mathrm{usage}(T)}{\max\limits_{T'}\{\mathrm{usage}(T')\}}. \tag{1}$$

For the state transition T due to handling sensory datum D at state S, DUC is defined as the ratio of the datum's usage against the maximal usage of any state transition T' due to handling any sensory datum D' at state S', which means that we compare all transitions based on their sensory data usage numbers and select the maximal one as T'. usage(T) is the number of valuable instructions, which produce observable benefits to users (e.g., GUI changes at forefront) and are executed when handling sensory datum D at state S, and a higher usage(T) value means more practical benefits. A low DUC value means poor sensory data utilization of sensory datum D at state S, as compared to others, which also implies that sensory datum D does not bring satisfactory benefits to users.

Finally, we rank all state transitions based on their DUC values, and present results to app developers. The result reports also contain event sequences for energy inefficiency scenario replay, and include information about how specific handlers use sensory data and what operations they have invoked. For low DUC values and their associated transitions, developers can carefully think about whether they are reasonable and necessary. If not, developers should consider how to optimize and improve concerned apps' design. Since our app state concept differs from GreenDroid, we calculate sensory data utilization coefficient with respect to transitions instead of states.

5 Evaluation

In this section, we experimentally evaluate our CyanDroid and compare it with GreenDroid.

5.1 Experimental setup and design

We implemented our approach as a prototype tool named CyanDroid on top of Java PathFinder (JPF)[4]. Figure 5 overviews our CyanDroid's system architecture. CyanDroid takes an Android app as input and conducts automated energy inefficiency diagnosis. It reports all detected energy inefficiency bugs after its app state exploration. Conceptually, CyanDroid consists of three major components: (1) `Android app executor` (AAE), which systematically generates event sequences to exercise an Android app; (2) `Data generator` (DG), which generates sensory data using multidimensional white-box sampling to explore different app states; (3) `Data utilization analyzer` (DUA), which tracks how the app transforms and utilizes sensory data at different states to identify those problematic states where sensory data are underutilized.

AAE parses all GUI elements of activities of an app and generates corresponding events. Besides, AAE simulates pressing button "Home" button and "Back". The mocked sensor events are also produced by AAE and fed to handlers. DG is implemented to collect hash index for each execution of a handler and generate new sample data delivered to AAE when necessary. We followed the GreenDroid's way to monitor

4) Java path finder. http://babelfish.arc.nasa.gov/trac/jpf/wiki.

Table 1 The basic information of experimental subjects

Apps	Rev No.	LOC	Source code availability
GPSLogger[a)]	R-15	659	Google code
Omnidroid[b)]	R-863	12427	Google code
Osmdroid[c)]	R-750	18091	GitHub
GeohashDroid[d)]	V-0.8.1-pre2	6682	Google code

a) GPSLogger. https://code.google.com/p/gpslogger/.
b) Omnidroid. https://code.google.com/p/omnidroid/.
c) Osmdroid. https://github.com/osmdroid/osmdroid.
d) GeohashDroid. https://code.google.com/p/geohashdroid/.

whether a handler is processing any sensor event. If it is, DUA further records valuable instructions which are related to sensory data to calculate sensory data usages of state transitions. Finally, AAE sorts and reports sensory data utilization coefficient for each state transition with corresponding triggering event.

To evaluate CyanDroid, we selected four open-source Android apps from GitHub and Google Code. We give their basic information in Table 1. They all heavily use sensors. Our evaluation is based on mutant testing [8]. Specifically, we randomly mutated our subjects to inject sensory data underutilization energy bugs into them. Such mutants have been proven to be valid substitutes for real bugs in software testing [9]. We explain how to automatically generate such mutants later. Our evaluation aims to answer the following three research questions:

- RQ1: Is GreenDroid's energy inefficiency diagnosis stable (always returning the same results)?
- RQ2: Can CyanDroid detect energy bugs (real or injected)? How does it compare to GreenDroid?
- RQ3: How does CyanDroid compare to GreenDroid in efficiency?

Besides the above research questions, we also conducted a case study of real apps to evaluate CyanDroid's bug detection capability. Our experiments were conducted on a machine with Intel Core i5 CPU @2.83 GHz and 4 GB RAM. The machine was installed with MS Windows 7 Professional.

5.2 Mutants generation

We mutated selected Android apps to obtain more experimental subjects. Unlike traditional mutation, we need to simulate energy bugs related to sensory data underutilization. Specifically, we mutated assignment and branching statements related to sensory data usage with creation, transformation and removal operations to change an app's sensory data processing flow. Details are as follows.

First, we retrieved methods that take sensor event objects as parameters (e.g., `onSensorChanged` and `onLoctionChanged`) from an app's source code. These methods are sensory event handlers. Then, we identified sensory data variables in these handlers' owner classes (e.g., `longitude`, `latitude` and `accuracy` for a typical location event handler). Next, we randomly generated arithmetic and conditional expressions of these sensory data variables, and mutated assignment and branching statements concerning sensory data in the retrieved methods. We deleted or replaced part of an assignment statement. For a branching statement, we performed one of the three operations randomly: (1) removing its condition predicate (becoming a tautology), (2) replacing its condition predicate with a new one, and (3) adding a new branch. For (2) and (3), their required code snippets were randomly selected from the app's program.

We generated 287, 82, 135 and 255 mutants for the four apps, respectively. Totally, 522 of them passed compilation and became our experimental subjects.

5.3 Instability of GreenDroid

Research question RQ1 concerns GreenDroid's instability in its analysis. We ran GreenDroid 10 times for each mutant and Figure 6 gives the distribution of mutants based on the number of different analysis results. The x-coordinate is the number of different results and the y-coordinate presents the corresponding number of mutants. For a mutant, a larger number of different results means more instable analysis. Figure 6 shows that that only 69 (13.2%) mutants have one constant result, while around 55.2% mutants have at least four kinds of different results.

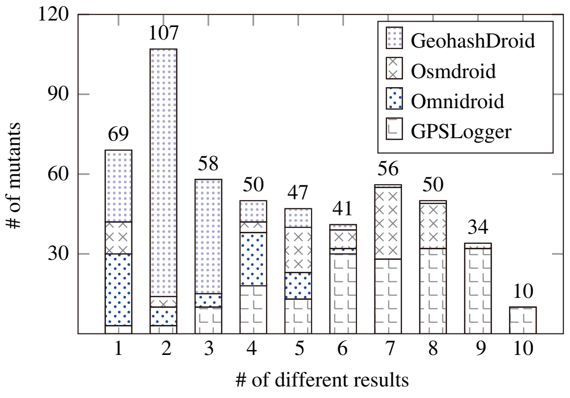

Figure 6 (Color online) Distribution of mutants in GreenDroid's analysis results.

Additionally, we calculated the percentage difference of transitions with DUC less than 0.5 for each mutant. The percentage difference is defined as follows:

$$\text{difference} = \max_{1\leqslant i\leqslant 10} |\overline{\text{Perc}} - \text{Perc}_i|, \tag{2}$$

where Perc_i is the percentage of transitions whose DUC are less than 0.5 for i-th analysis and $\overline{\text{Perc}}$ is the average percentage for 10 times analysis. Therefore, a percentage difference of 0.0 implies a perfect stable analysis result, while the larger the difference is, the more instable the analysis result is. Our calculation discloses that the maximal percentage difference is 0.82, 0.37, 0.77 and 0.73, for the four subjects, respectively, while their averaged percentage differences are 0.48, 0.08, 0.47 and 0.02.

Both of the number of different results and the percentage difference suggest the instability of GreenDroid in terms of analyzing one subject as well as analyzing across different subjects. We also observe that GreenDroid missed problematic states where sensory data were underutilized (when reporting different analysis results), thus missing potential bug reports. GreenDroid's instability might confuse developers by returning different results each time. Our CyanDroid does not have this problem since its sensory data generation is deterministic and systematic, and its analysis result is combined from all possible executions.

5.4 Effectiveness of CyanDroid

A CyanDroid's analysis report contains analyzed state transitions and their DUC values, which indicate their sensory data utilization levels. Since many state transitions share the same DUC values, they are grouped together rather than being listed separately. For an analysis report, a larger number of such groups (representing different DUC values) is considered to be a more thorough analysis. For example, a GreenDroid's analysis report can contain three different DUC values for an app subject, while a CyanDroid's analysis report may contain five different DUC values for this subject, representing that CyanDroid has explored more use scenarios than GreenDroid.

To answer research question RQ2, we compared the analysis results of CyanDroid and of GreenDroid. For each mutant, we applied GreenDroid to analyze it for 10 times and chose its best analysis result result for comparison with Cyndroid. Here, "best" means that the analysis report contains the most number of different DUC values among 10 candidate reports, and in the following we call this best analysis result GreenDroid's analysis result for simplicity. Table 2 gives the detailed comparison.

For ease of presentation, we name a state transition (i.e., handler execution) whose DUC value is no more than 0.3 as bug transition (whose sensory data utilization is significantly low), transition ($0.3 <$ DUC $\leqslant 0.7$) hotspot transition (whose sensory data utilization is somewhat low), and transition (DUC $>$ 0.7) norm transition (whose sensory data utilization is just fine). For each app, a row named "Positive" indicates that CyanDroid has an improved analysis report as compared to GreenDroid. Here, "improved" means that CyanDroid has an analysis report containing more different DUC values than GreenDroid.

Table 2 Effectiveness comparisons between CyanDroid and GreenDroid

App	Comparison	Bug transition DUC $\leqslant 0.3$		Hotspot transition $0.3 < \text{DUC} \leqslant 0.7$		Norm transition $0.7 < \text{DUC}$	
		Perc (%)	More (%)	Perc (%)	More (%)	Perc (%)	More (%)
GPSLogger	Positive	35.2	100.0	12.3	100.0	10.6	100.0
	Neutral	64.8	–	86.0	–	89.4	–
	Negative	0.0	–	1.7	−50.0	0.0	–
Omnidroid	Positive	14.1	130.0	12.7	92.6	12.7	92.6
	Neutral	83.1	–	85.9	–	85.9	–
	Negative	2.8	−41.7	1.4	−50.0	1.4	−50.0
Osmdroid	Positive	38.9	95.7	82.2	58.2	24.4	81.8
	Neutral	60.0	–	16.7	–	75.6	–
	Negative	1.1	−100.0	1.1	−33.3	0.0	–
Geohash Droid	Positive	3.3	100.0	6.0	100.0	3.3	100.0
	Neutral	96.2	–	94.0	–	96.7	–
	Negative	0.5	−100.0	0.0	–	0.0	–

Similarly, “Negative” and “Neutral” means that CyanDroid’s analysis report is worse than and the same as GreenDroid’s report in quality (i.e., number of different DUC values), respectively. Note that we compare CyanDroid’s and GreenDroid’s analysis reports three times, with respect to three different types of state transitions (bug, hotspot and norm transitions), respectively. For example, a block fenced by “Bug transition” and “Positive” in Table 2 means the measurement of how many mutants have an analysis report that have more different DUC values within the range of $[0, 0.3]$ reported by CyanDroid as against GreenDroid. The measurement contains “Perc” and “More” two parts. The former reports the percentage of mutants meeting this measurement’s goal against the total number of mutants. The latter reports the average increased percentage of different DUC values (CyanDroid vs. GreenDroid) among such mutants. Note that for “Negative” cases, this value is negative (as it was actually decreased) or “–” (i.e., not applicable when there is no such case).

Bug and hotspot transitions are our main concerns since they relate to very low sensory data utilization. We observe from Table 2 (“Perc” column for “Bug transition”) that “Positive” cells contain much larger values than “Negative” cells in most cases. For GeohashDroid, their difference is not large (only 2.8%), but for this case, “Neutral” cells contain significantly large values (over 96%). This shows that CyanDroid can detect serious energy bugs in a much more effective way than GreenDroid by exposing much more use scenarios where sensory data are significantly underutilized (DUC $\leqslant 0.3$). For hotspot transitions (“Perc” column), “Positive” cells all contain much larger values than “Negative” cells, and their differences vary from 6.0% to 81.1%. Besides, for the “More” column (all three transition types), we observe that “Positive” cells almost all contain much larger values than “Negative” cells. These results suggest that CyanDroid can detect more energy bugs than GreenDroid in most cases, although there are few cases where GreenDroid performs better than CyanDroid, whose chance is, however, very low (0–2.8%).

We owe CyanDroid’s more effective energy bug analysis to its more thorough exploration of an app subject’s execution paths. To confirm it, we compared CyanDroid’s analyzed paths for each mutant’s sensory event handler against GreenDroid. Note that we did not use path coverage here for comparison since it can be infinite when an analyzed sensory event handler contains any loop. For comparison, we recorded different paths in a sensory event handler that were executed in analysis by the code dynamic monitoring functionality from JPF. We did not instrument any code into the apps under analysis, which may disturb the diagnosis of CyanDroid. Instead, we added listeners to JPF when implementing CyanDroid on top of JPF to record executed instructions for inferring concrete execution paths. We present comparison results in Table 3. Still, for GreenDroid, we ran it 10 times for each mutant and averaged their results.

Column “Positive” means that CyanDroid covered more paths than GreenDroid. Columns “Negative” and “Neutral” indicate that CyanDroid covered fewer than and the same number of paths as GreenDroid,

Table 3 Path comparisons between CyanDroid and GreenDroid

App	Positive		Neutral	Negative	
	Perc (%)	More (%)	Perc (%)	Perc (%)	More (%)
GPSLogger	88.3	67.8	10.1	1.7	−17.8
Omnidroid	28.2	152.3	69.0	2.8	−32.1
Osmdroid	57.8	62.3	42.2	0.0	–
GeohashDroid	99.5	121.7	0.5	0.0	–

Table 4 Average analysis overhead comparisons

App	Number of states		Time (s)		Memory (MB)	
	GreenDroid	CyanDroid	GreenDroid	CyanDroid	GreedDroid	CyanDroid
GPSLogger	1332	91694	39	891	21	24
Omnidroid	2191	7394	754	1223	44	60
Osmdroid	4297	125161	57	967	36	43
GeohashDroid	910	160472	19	2424	26	32

respectively. Similar to previous comparisons, the three “Perc” columns measure the percentage of mutants for “Positive”, “Negative” and “Neutral” three cases, respectively. The two “More” columns measure how many more program paths CyanDroid covered than GreenDroid in percentage, averaged over mutants for “Positive” and “Negative” two cases, respectively. As we explained earlier, “More” may have negative values (for “Negative” cases, where CyanDroid covered less paths by definition). As we can observe from the Table 3, “Positive” mutants are the majority and their “Perc” values range from 28.2% to 99.5% with an average of 68.45%, which is significant, while the largest “Perc” value for “Negative” mutants is only 2.8%. Besides, the “More” values of “Positive” mutants are clearly much larger than those of “Negative” mutants, and their and their average values are 101.3% and −12.5%, respectively. These results strongly suggest that CyanDroid can indeed explore much more program paths in app analysis than GreenDroid.

Overall, CyanDroid not only covered more program paths during analysis, but also traversed more app states. Nevertheless, there are still few subjects and cases where CyanDroid might report slightly worse results than GreenDroid. The reason is that the random strategy in GreenDroid might generate lucky sensory data that happen to exercise paths that cannot be covered by CyanDroid, but such chances are low as we observe from experiments. In general, these comparison results confirm the effectiveness of our CyanDroid in detecting energy inefficiency bugs caused by sensory data underutilization.

5.5 Efficiency of CyanDroid

We then consider research question RQ3 that concerns CyanDroid’s efficiency. We measured CyanDroid’s analysis overhead and compared it to that of GreenDroid (averaged over 10 runs) in Table 4. CyanDroid aims to examine more execution paths for each event handler. Thus it typically explores more application states than GreenDroid. As shown in Table 4, CyanDroid explored 3.3x−176.3x application states as many as GreenDroid on our experimental subjects. However, the time overhead of CyanDroid did not increase that significantly. We observe that GreenDroid’s analysis time ranges from 39 to 754 s, while CyanDroid’s ranges from 891 to 2424 s, which are larger. The ratios of the analysis time between CyanDroid and GreenDroid are 22.8, 1.6, 17.0 and 127.6, respectively, for the four experimental subjects. Although CyanDroid worked slower than GreenDroid for these app subjects, CyanDroid obtained stable analysis results after only one run, while GreenDroid did not obtain stable analysis results even after 10 runs. In fact, these 10 runs may take more time than CyanDroid (i.e., even slower and even worse results). Therefore, we consider that CyanDroid is still competitive as compared to GreenDroid in the analysis efficiency. Besides, CyanDroid incurred slightly more memory cost than GreenDroid, since it explored and analyzed much more program paths.

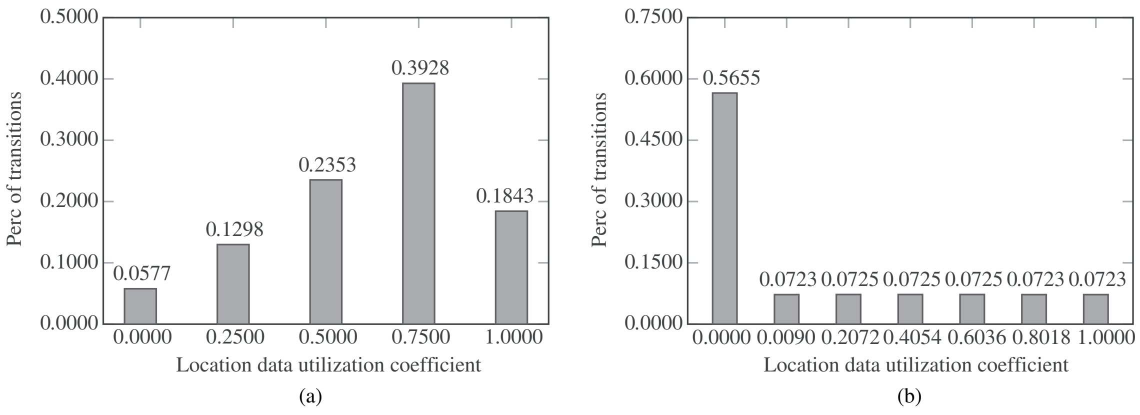

Figure 7 Analysis results of case study. (a) GPSLogger's analysis report by CyanDroid; (b) Omnidroid's analysis report by CyanDroid.

5.6 Case study

Previously, we evaluated our CyanDroid's effectiveness and efficiency in analyzing sensory data under-utilization issues for Android apps using their mutants. In the following, we report our analysis results for two real-world Android apps using their original programs.

5.6.1 *GPSLogger*

GPSLogger is an Android app that helps users label their photos with GPS locations and record their daily routes. Figure 7(a) gives CyanDroid's analysis result of GPSLogger, which shows that almost half of its analyzed state transitions are problematic with 0.0, 0.25 or 0.5 such low DUC values. By looking into GPSLogger's source code, we found that GPSLogger discards sensed location data directly if the data do not meet certain precision requirements. If users stay in some places with bad GPS signals, the app may keep discarding the obtained data. Then it is meaningless to use the app for location recording and much battery energy will be wasted. So in order to optimize for the app's energy efficiency, developers should considering reducing the location sensing frequency or temporarily disabling the sensing when the app keeps receiving imprecise location data for some long time.

5.6.2 *Omnidroid*

The app Omnidroid helps users automatically perform some tasks in certain contexts with predefined rules. For example, if a user receives a phone call when she/he is having a meeting, Omnidroid can help automatically reply a message with the content like "I am busy". From the analysis result in Figure 7(b), we can observe that there are quite many state transitions with a DUC value of zero, which undoubtedly indicates its serious energy. Besides, there are also many cases where DUC values are clearly low and range in a large scope (0.0090–0.8018). We replayed these problematic cases, studied the corresponding code and realized the responsible problem. If there are rules concerning location data, Omnidroid will listen to location update events. Every time when there is an update, it will construct an `OmniArea` object with the received location data. However, the `longitude` and `latitude` fields of the object are mistakenly switched as shown in Figure 8. Therefore, if `longitude` is greater than 90 or less than -90, the constructor of `OmniArea` will throw an exception (the range of latitude is -90 to 90) and just returns. This exception will be caught but ignored, and therefore the location data will not be utilized. If users stay in the area where the `longitude` is greater than 90 or less than -90, the app Omnidroid may repeat this useless process for many times if it continuously listens to location updates. This can cause Omnidroid to waste a large amount of battery energy. This problem is found by CyanDroid and previously unknown. We have reported it to OmniDroid's developers[5].

5) Issue 189 - Omnidroid. https://code.google.com/p/omnidroid/issues/detail?id=189.

```java
public void onLocationChanged(Location location) {
    OmniArea newLocation;
    try {
        // BUG: the longitude and latitude are mistakenly switched
        newLocation = new OmniArea(null, location.getLatitude(),
            location.getLongitude(), location.getAccuracy());
    } catch (DataTypeValidationException e) {
        newLocation = null;
    }
    //follow-up processing
}
```

(a)

```java
public OmniArea(String userInput, double longitude, double latitude,
    double proximityDistance) throws DataTypeValidationException {
    if (latitude < MIN_LATITUDE || latitude > MAX_LATITUDE)
        throw new DataTypeValidationException("Latitude must be between "
            + MIN_LATITUDE + " and " + MAX_LATITUDE);
    if (longitude < MIN_LONGITUDE || longitude > MAX_LONGITUDE)
        throw new DataTypeValidationException("Longitude must be between "
            + MIN_LONGITUDE + " and " + MAX_LONGITUDE);
    if (proximityDistance < 0)
        throw new DataTypeValidationException("Distance cannot be negative");
    //follow-up processing
}
```

(b)

Figure 8 (Color online) Omnidroid's code snippets. (a) OmniArea.java; (b) LocationManager.java.

5.7 Discussion

Our CyanDroid is effective in analyzing sensory data underutilization issues, but it may suffer from the state space explosion problem. The reason is that CyanDroid splits a state space always in the middle for each dimension, which imposes a logarithmic complexity in a dimension, and stops search immediately after it reaches the threshold associated with that dimension. This treatment can introduce more states explored than in GreenDroid, thus leading to increased time and memory cost. However, the time overhead of CyanDroid does not increase that significantly. Besides, CyanDroid only needs to run once, while GreenDroid may have to run many times to make its users feel sure about which results are stable. We plan to alleviate this problem by exploiting existing pruning techniques to reduce redundant states [10]. We also plan to further study this problem for the case where there are many dimensions of sensory data in future.

Besides, the current CyanDroid implementation would generate random strings when an Android app needs text inputs. As such, CyanDroid cannot simulate complicated user events, such as screen unlock event, valid user name and password. This simplifies the implementation but may cause some app states not reachable. Therefore, CyanDroid needs extension to incorporate other program analysis techniques [11] to better explore an app's state space. This requires our further effort.

In the experiments, we concerned only location-aware Android apps because they are popular nowadays (most Android apps use or rely on location data directly or indirectly). Our CyanDroid approach does not make any assumption on the types of sensory data for analysis. In terms of tool implementation, currently our CyanDroid supports multidimensional location data. We are also extending it to support more types of sensory data and the extension does not change our sensory data analysis.

Finally, our evaluation involves large-scale experiments with mutants but relatively small-scale case studies with real-world Android apps. A major reason is that CyanDroid needs quite extension to handle implementation issues with every specific app's customized exceptions as well as native API calls. We plan to work along this line to test more Android apps in future.

6 Related work

Due to the popularity of smartphone uses with ordinary people, researchers have spent great efforts on assuring both functional and non-functional quality properties for smartphone apps. Some work focuses on quality assurance via functional testing. For example, Dynodroid generated relevant inputs for automated Android app testing [12]. EvoDroid proposed a segmented evolutionary technique for testing Android apps in a more effective way [13]. UGA leveraged human insights for semi-automated testing of Android apps, and proved to be more effective than pure random testing techniques or strategies [11]. For non-functional testing, one good example is the static analysis tool PerfChecker that detects performance bugs in Android apps by summarized patterns, e.g., view holder pattern [14]. In this article, we mainly concern energy inefficiency bugs for smartphone apps.

Researchers have proposed various techniques to detect energy inefficiency bugs in smartphone apps. For example, Pathak et al. [15] characterized no-sleep energy bugs in Android apps and then used reaching-definition data-flow analysis to detect their occurrences. However, this work cannot detect energy inefficiency bugs related to low sensory data utilization as we studied in this article. Besides, it requires information about all calling sequences among handlers, which cannot be easily available in practice. Liu et al. [2, 3] proposed GreenDroid to automatically execute Android apps and search problematic states with low sensory data utilization by dynamic tainting, as our focus in this article. However, as we discussed earlier, they may miss critical states that are reachable only by specific sensory data inputs and thus diagnose apps in an incomplete and unstable way. Banerjee et al. [16] presented an automated test generation framework that diagnoses energy bugs/hotspots in Android apps, but the work required hardware support for real measurement.

There is also existing work concerning energy consumption estimation. Singh et al. [17] used hardware instrumentation to estimate actual power usage for each computing task at an OS level. Pathak et al. [18] implemented Eprof to help measure energy consumption of an Android apps from collected event sequences. Hao et al. [19] and Li et al. [20] proposed calculating fine-grained energy consumption information at a source-line level. This is done by combining hardware-based power estimation and statistical modeling. These pieces of work all require concrete hardware platforms for app execution, which our work in this article does not need.

Some other pieces of work concerns empirical studies or experiments about energy consumption for smartphone apps. Sahin et al. [21] conducted an empirical study to investigate the impact of code obfuscation on energy usage for Android apps. They found that the impact can be marginal to app users. Li et al. [22] presented an empirical investigation to analyze energy consumption of real-world Android apps at a source-code level. They found that the idle state can drain most of energy while a few certain APIs including network ones dominate non-idle energy consumption. Linares-Vasquez et al. [23] presented their large quantitative and qualitative empirical study into the categories of API calls and usage patterns that can exhibit particularly high energy consumption profiles. These pieces of work focus on finding out characteristics of smartphone app energy inefficiency, and our work detects such issues concretely. Some observations echo with each other, and we believe that more observations can be incorporated for diagnosis of more energy inefficiency bugs in future.

Additionally, some work also helps developers fix energy inefficiency bugs. Ma et al. [24] proposed a practical tool eDoctor that helps developers diagnose abnormal battery drain issues in apps and also presented some feasible solutions for fixing them. Li et al. [25] instrumented apps that have energy inefficiency bugs to automatically fix sensor data underutilization issues, and their app-specific energy-aware sensing policies can be integrated with the apps without any modification to the underlying operating systems or the apps themselves.

Finally, white-box sampling is similar to Non-uniform sampling in signal processing, which accurately selects some points to reconstruct the signal in the receiving end as soon as possible [26]. Non-uniform sampling needs to sample more data points in a place where energy changes rapidly, which is like the case that white-box sampling generates more data points at discrete-decision points in paths. However, the main difference is that white-box sampling generates sampled data for an interval based on hash indexes of historical executions while non-uniform sampling works under predefined non-uniform sample interval, which is also only one dimension, with statistical and pseudo-random methods.

7 Conclusion

In this article, we studied the energy inefficiency diagnosis problem for smartphone apps. We focused on its stability and effectiveness aspects, as motivated by limitations of a recent state-of-the-art technique, GreedDroid. We proposed the fine-granularity state concept and multidimensional white-box sampling technique to fulfill the goal of systematic exploration of an app's state space and relate it to different usage scenarios of sensory data. Our approach can explore an app's state space automatically, and as the evaluation shows, its tool implementation, CyanDroid, exhibited its stability, effectiveness and efficiency in smartphone app energy inefficiency diagnosis. Our CyanDroid also reported real bugs in real-world apps. In future, we plan to validate our idea in more real-world apps, and study dynamic adaptation of input space splitting thresholds to automatically balance time cost and diagnosis effectiveness.

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

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Carroll A, Heiser G. An analysis of power consumption in a smartphone. In: Proceedings of the USENIX Annual Technical Conference, Boston, 2010. 1–14

2 Liu Y P, Xu C, Cheung S C. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In: Proceedings of IEEE International Conference on Pervasive Computing and Communications (PerCom), San Diego, 2013. 2–10

3 Liu Y P, Xu C, Cheung S C, et al. GreenDroid: automated diagnosis of energy inefficiency for smartphone applications. IEEE Trans Softw Eng, 2014, 40: 911–940

4 Bao T, Zheng Y H, Zhang X Y. White box sampling in uncertain data processing enabled by program analysis. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, Tucson, 2012. 897–914

5 Sama M, Rosenblum D S, Wang Z M, et al. Model-based fault detection in context-aware adaptive applications. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, 2008. 261–271

6 King J C. Symbolic execution and program testing. Commun ACM, 1976, 17: 385–394

7 Kemerlis V P, Portokalidis G, Jee K, et al. Libdft: practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, London, 2012. 121–132

8 Jia Y, Harman M. An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng, 2011, 37: 649–678

9 Just R, Jalali D, Inozemtseva L, et al. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, 2014. 654–665

10 Knuth D E, Moore R W. An analysis of alpha-beta pruning. Artif Intell, 1976, 6: 293–326

11 Li X J, Jiang Y Y, Liu Y P, et al. User guided automation for testing mobile apps. In: Proceedings of Asia-Pacific Software Engineering Conference, Jeju, 2014. 27–34

12 Machiry A, Tahiliani R, Naik M. Dynodroid: an input generation system for Android apps. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, 2013. 224–234

13 Mahmood R, Mirzaei N, Malek S. EvoDroid: segmented evolutionary testing of Android apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, 2014. 599–609

14 Liu Y P, Xu C, Cheung S C. Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 36th International Conference on Software Engineering, Hyderabad, 2014. 1013–1024

15 Pathak A, Jindal A, Hu Y C, et al. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services. New York: ACM, 2012. 267–280

16 Banerjee A, Chong L K, Chattopadhyay S, et al. Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, 2014. 588–598

17 Singh D, Kaiser W J. The atom LEAP platform for energy-efficient embedded computing. Technical Report, University of California, Los Angeles. 2010

18 Pathak A, Hu Y C, Zhang M. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with eprof. In: Proceedings of the 7th ACM European Conference on Computer Systems, Bern, 2012. 29–42

19 Hao S, Li D, Halfond W G J, et al. Estimating mobile application energy consumption using program analysis. In: Proceedings of the International Conference on Software Engineering, San Francisco, 2013. 92–101

20 Li D, Hao S, Halfond W G, et al. Calculating source line level energy information for Android applications. In: Proceedings of the International Symposium on Software Testing and Analysis, Lugano, 2013. 78–89

21 Sahin C, Tornquist P, McKenna R, et al. How does code obfuscation impact energy usage? In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, 2014. 131–140

22 Li D, Hao S, Gui J P, et al. An empirical study of the energy consumption of Android applications. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, 2014. 121–130

23 Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, et al. Mining energy-greedy API usage patterns in Android apps: an empirical study. In: Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, 2014. 2–11

24 Ma X, Huang P, Jin X X, et al. eDoctor: automatically diagnosing abnormal battery drain issues on smartphones. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, Lombard, 2013. 57–70

25 Li Y C, Guo Y, Kong J J, et al. Fixing sensor-related energy bugs through automated sensing policy instrumentation. In: Proceedings of IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), Rome, 2015. 321–326

26 Marvasti F. Nonuniform Sampling: Theory and Practice. New York: Springer Science & Business Media, 2012