SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

NavyDroid: An Efficient Tool of Energy Inefficiency Problem Diagnosis for Android Applications

Yi LIU[1,2], Jue WANG[1,2], Chang XU[1,2*], Xiaoxing MA[1,2] & Jian LU[1,2]

[1]*State Key Lab for Novel Software Technology, Nanjing University, Nanjing* 210023*, China;*
[2]*Department of Computer Science and Technology, Nanjing University, Nanjing* 210023*, China*

Abstract Energy inefficiency is an influential non-functional issue for smartphone applications, causing increased concerns from users. Locating these problems is labor-intensive, thus automated diagnosis tools are in demand. Some existing approaches detect energy inefficiency problems by exploring application states with the JPF framework, and get favorable results. However, the effects of these approaches are restricted because of their imprecise application execution models and incomplete energy inefficiency patterns. This article introduces NavyDroid, an effective and efficient tool of energy inefficiency problem diagnosis for Android applications. We constructed a comprehensive application execution model in the form of a state machine, which accurately simulates the runtime behavior of Android applications. We designed a parallel algorithm to systematically explore an application's state space. Our approach supports more energy inefficiency patterns, and is able to detect complicated wake lock misuses. We implemented our approach as a prototype tool and applied it to real-world applications. We evaluated NavyDroid with 19 real-world Android applications, and NavyDroid located more energy inefficiency bugs in these applications than the existing work E-GreenDroid did. Also, NavyDroid reduced the analysis time with its parallel state exploration algorithm. The experimental results demonstrated the effectiveness and efficiency of our approach for detecting energy inefficiency bugs in Android applications.

Keywords energy inefficiency, smartphone application, wake lock

Citation Yi Liu, Jue Wang, Chang Xu, Xiaoxing Ma and Jian Lu. NavyDroid: An Efficient Tool of Energy Inefficiency Problem Diagnosis for Android Applications. Sci China Inf Sci, for review

1 Introduction

Nowadays, as smartphones become more and more popular, the number of Android applications is growing rapidly. The data show that there are lots of applications and cumulative downloads on the Google Play Store [7]. However, many applications suffer from energy inefficiency problems. These applications employ energy-consuming operations, such as location sensing, to provide a good user experience. At the same time, Android developers are responsible for managing the power of the device. Therefore, if these energy-consuming operations are not performed properly, much energy will be wasted. In this case, the device battery can be exhausted in a few hours, and this will result in user frustration and complaints. As energy inefficiency problems become more common, users become concerned about this issue.

It is difficult for developers to diagnose energy inefficiency problems. These problems occur only at certain application states. In order to reproduce the problems, developers often need to explore a variety of application states. Therefore, automatic detection tools help locate the problems and increase debugging efficiency.

* Corresponding author (email: changxu@nju.edu.cn)

In past several years, researchers have proposed different tools to detect energy inefficiency problems automatically. Pathak et al. conducted the first study in the area of detecting energy bugs for smartphone applications [30]. Zhang et al. proposed ADEL to detect energy leaks of network data [34]. These approaches employ different analysis techniques. Among them, several pieces of work simulate the execution of applications upon a verification framework JPF, and are shown to be effective [20, 23, 32]. GreenDroid [23] detects the missing releasing of sensors and wake locks, and analyzes whether sensor data are effectively utilized. CyanDroid [20] systematically generates multidimensional sensor data to reproduce bugs that require specific sensory data to manifest. E-GreenDroid [32] optimizes the simulation execution, and updates the library modeling of GreenDroid, including some new features of the Android system.

Generally, these energy inefficiency detection tools are composed of two parts, namely, the *simulation* part and the *monitor* part.

1. **Simulation.** The *application execution engine* simulates the execution of Android applications, and explores application states. The two main processes of the simulation part are event sequence generation and state space exploration. Both processes are guided by the *application execution model*.

2. **Monitor.** The monitor part is based on the simulation part, and operates during the simulation execution. It monitors suspicious operations of the application, and checks for operations that match *energy inefficiency patterns*. An investigation shows that many energy inefficiency problems associate with two types of *energy inefficiency patterns* [23]:

   • **Missing sensor listener or wake lock deactivation.** An application registers a sensor listener to fetch sensor data, and acquires a wake lock to keep the CPU awake for long background tasks. The sensor listener should be unregistered when the senor is no longer needed, and the wake lock should be released when background tasks are done. If sensor listeners or wake locks are not deactivated in time, the battery power can be exhausted rapidly. [4, 5, 11].

   • **Sensor data underutilization.** The sensor itself consumes energy to retrieve sensor data. Thus, applications should utilize sensor data in an effective way. Applications using sensor data inefficiently can be viewed as a waste of energy.

Existing approaches that simulate the execution of an Android application follow the above two-part architecture. However, these approaches have some limitations in both parts. As for the simulation part, the application execution model is imprecise. As for the monitor part, the energy inefficiency patterns are not complete. Here we discuss these shortcomings in detail and propose our solution.

In the simulation part, the application execution model guides the simulation execution of applications. However, due to the complicated component lifecycle and runtime behavior of Android applications, the behavior guided by the model is different from the actual behavior. Concretely, the models in existing approaches are not accurate in the lifecycle of activities. They do not include paused and killed states of activities.

Meanwhile, existing approaches have poor performance in exploring applications' states. E-GreenDroid randomly generates event sequences for state exploration. However, in this way duplicated event sequences are generated, and the exploration is not efficient. There should be a better approach to exploring application states. Also, existing approaches to generating event sequences can cause an application to quit abnormally, resulting in false positives.

Also, existing work did not summerize the complete energy inefficiency patterns in the monitor part. For instance, the status of a wake lock is not as simple as acquired and released. With reference counts, wake locks have more complicated misuse patterns, e.g., multiple lock acquisitions. Existing approaches do not check for these complicated misuse patterns.

Therefore, in order to address the above issues, we propose our approach in this article. We define an application execution model that precisely simulates the lifecycle of Android components. The model

is constructed in the form of a strengthened deterministic finite automaton (DFA). The automaton contains the paused state and the killed state of an activity, as well as their related state transitions. We improve the process of event sequence generation to ensure that an application terminates normally in simulation execution. We design a state exploration algorithm to systematically explore the state space of applications, and parallelize it to accelerate the exploration. Also, we summerize new misuse pattern detecting policies for the monitor part.

We implemented our approach as a prototype tool named NavyDroid for evaluation. We selected 19 real-world Android appliations in order to evaluate the effectiveness and efficiency of NavyDroid in energy inefficiency diagnosis. We compare NavyDroid with E-GreenDroid [32], a state-of-the-art energy inefficiency detection tool. We analyzed the test subjects with both tools, and compared their analysis reports. As a result, NavyDroid located more energy inefficiency bugs than E-GreenDroid did, and located all the energy inefficiency bugs that E-GreenDroid reported. Moreover, we evaluated the efficiency of our parallel state exploration algorithm. The results demonstrate that it takes average 40% less time to explore application states compared with the random exploration used in E-GreenDroid, while preserving the same effectiveness. We can conclude from the evaluation results that NavyDroid detects energy inefficiency problems more effectively and efficiently.

In summary, our work makes the following contributions:

- We design an algorithm to systematically explore application states. We make the algorithm more efficient by parallelizing its execution.
- We construct an accurate application execution model. The model supports more activity states, and is more consistent with the lifecycle of Android components.
- We enhance the monitor part to detect more wake lock misuse patterns. Our approach supports more energy inefficiency patterns such as multiple lock acquisitions.
- We implement our approach as a prototype tool named NavyDroid and evaluate it with real-world Android applications. NavyDroid is able to detect more energy inefficiency bugs in the test subjects, indicating its effectiveness. Also, NavyDroid takes average 40% less time for state exploration compared with E-GreenDroid.

The work presented in this article is based on our previous work [26], and has significantly extended it. Previously, we used the same random event sequence generation as E-GreenDroid. However, this approach generates many duplicated event sequences, and shows poor efficiency. In this work, we have extended our approach with a state exploration algorithm, which can systematically explore an application's state space in analysis and eliminate repeated event sequences. Moreover, we parallelize the exploration algorithm to accelerate the execution. Evaluation results show that the parallel algorithm takes average 40% less time compared with the original random exploration.

The rest of this article is organized as follows. Section 2 introduces some background of Android programming and presents a motivating example. Section 3 elaborates on our approach to detecting energy inefficiency. Section 4 evaluates NavyDroid with real-world applications. Section 5 and Section 6 discuss our work and some related work, and finally Section 7 concludes this article.

2 Background and Motivation

In this section, we introduce the basics of Android applications, and present a motivating example of our work.

2.1 Background

Application components are the essential parts of an Android application. There are four different types of application components [2]:

Activity. Activities are the only components for graphical user interfaces (GUI). The GUI layouts corresponding to activities are declared in configuration files. The entry point of an application is usually

```
1  public class PlaybackActivity extends Activity {
2      private ImageButton mPlayPauseButton;
3      private PlaybackService mPlaybackService;
4      private ServiceConnection mConnection = new ServiceConnection() {
5          public void onServiceConnected(
6                  ComponentName name, IBinder binder) {
7              mPlaybackService = ((MyBinder) binder).getService();
8          }
9      };
10     public void onCreate(Bundle state) {
11         mPlayPauseButton.setOnClickListener(new OnClickListener() {
12             public void onClick(View v) {
13                 // toggle play/pause state
14                 mPlaybackService.playPause();
15             }
16         });
17         Intent i = new Intent(this, PlaybackService.class);
18         // start PlaybackService
19         startService(i);
20     }
21     public void onResume() {
22         Intent i = new Intent(this, PlaybackService.class);
23         // bind PlaybackService
24         bindService(i, mConnection, Context.BIND_ABOVE_CLIENT);
25     }
26 }
27
28 public class PlaybackService extends Service
29         implements MediaPlayer.OnPreparedListener {
30     private static String TAG = PlaybackService.class.getName();
31     private WakeLock mWakeLock;
32     private MediaPlayer mMediaPlayer;
33     public void onCreate() {
34         mWakeLock = getPowerManager().newWakeLock(
35                 PowerManager.PARTIAL_WAKE_LOCK, TAG);
36         mMediaPlayer = createAndSetupMediaPlayer();
37         mMediaPlayer.setOnPreparedListener(this);
38     }
39     public void onDestroy() {
40         // stop media when the service is destroyed
41         stop();
42     }
43     public void onPrepared() {
44         // start media when the media player get prepared
45         start();
46     }
47     public void playPause() {
48         if (mMediaPlayer.isPlaying())
49             pause();
50         else
51             start();
52     }
53     public void start() {
54         // play media and acquire wake lock
55         mWakeLock.acquire();
56         mMediaPlayer.start();
57     }
58     public void stop() {
59         // stop media and release wake lock
60         mMediaPlayer.stop();
61         if (mWakeLock.isHeld()) mWakeLock.release();
62     }
63     public void pause() {
64         // pause media and release wake lock
65         mMediaPlayer.pause();
66         if (mWakeLock.isHeld()) mWakeLock.release();
67     }
68     public class MyBinder extends Binder {
69         PlaybackService getService() {
70             return PlaybackService.this;
71         }
72     }
73     public IBinder onBind(Intent intent) {
74         return new MyBinder();
75     }
76 }
```

Figure 1 Motivating example from the TomaHawk application (revision 543c3b9ab4)

an activity. During the execution, different running activities are organized in a *back stack*. The top activity of the back stack is called the activity at foreground.

Service. A service runs at the background to perform long-time tasks. An activity can start a service, or *bind* to a service and interact with it.

Broadcast receiver. A broadcast receiver is a component that responds to system-wide broadcast messages. A broadcast receiver can be another entry point into the application besides its own activities.

Content provider. A content provider manages a set of shared application data. Through the content provider, other components and applications can query or modify the data.

Each application component follows a prescribed lifecycle when it transits through different states [1]. As a component enters a new state, the Android framework invokes the corresponding event handlers. Scheduling these event handlers is essential to the simulation execution of an Android application.

2.2 Motivating Example

We present two energy inefficiency bugs in TomaHawk [13] as the motivating example. Figure 1 shows a simplified version of the problematic code snippet. The `PlaybackActivity` and `PlaybackService` play the media specified by users. Users can switch the mode of media playing by clicking the play/pause button (Lines 12–15, 48–51). `PlaybackService` is started by `PlaybackActivity` and runs at background (Lines 17–19). The activity binds to the service for interactions (Lines 22–24). In order to prevent the media playing from being interrupted, `PlaybackService` plays the media with a wake lock (Lines 34–35). When the media playing starts, the wake lock will be acquired (Lines 54–56); when the media playing pauses or stops, the wake lock will be released (Lines 59–61, 64–66).

There are several operations on one wake lock in `PlaybackService`. When the media player gets prepared, the wake lock will also be acquired (Lines 43–46). The service checks whether the wake lock is being held before releasing it (Lines 61, 66), but acquires the wake lock directly without checking (Line 55). Therefore, the wake lock can be acquired more than once. If a user clicks the play/pause button twice after the media player gets prepared, the wake lock will be acquired twice but released only once. As wake locks are reference-counted by default, the wake lock calculates its reference count as the number of acquires minus the number of releases. The wake lock will keep being held because its reference count

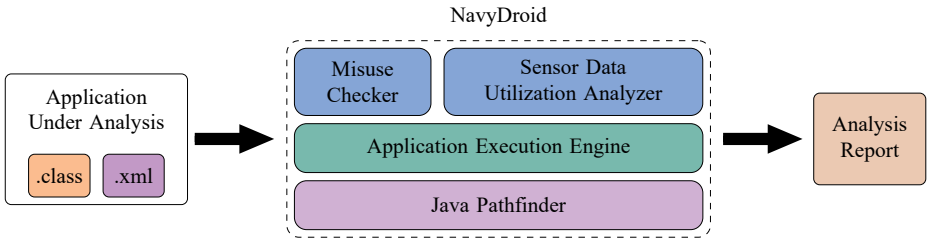


Figure 2 Approach overview

is larger than zero [12]. In this case, the working CPU will consume battery power without any user benefit. Existing tools, such as GreenDroid and E-GreenDroid, fail to detect this energy inefficiency problem, because of the incomplete wake lock misuse patterns. They cannot address this usage mode of wake locks.

Another energy inefficiency bug occurs when the activity and the service get killed by the Android system. When a user exits `PlaybackActivity`, both `onDestroy()` callbacks of the activity and the service will be invoked, and `PlaybackService` stops media playing and releases the wake lock (Lines 40–41). However, the Android system may kill processes when the memory is insufficient, in which case the `onDestroy()` callbacks of killed activities and services will not be invoked [1]. In this case, the wake lock will not be properly released, consuming the battery power. GreenDroid and E-GreenDroid, with imprecise application execution models, cannot simulate the runtime behavior of activities or services being killed. So they fail to detect this energy inefficiency problem.

The above example motivates us to propose an approach that can accurately simulate the execution of Android applications, and identify complex wake lock misuse patterns.

3 NavyDroid Approach

In this section, we elaborate on our approach to detecting energy inefficiency problems in Android applications.

3.1 Overview

NavyDroid follows the two-part architecture described in Section 1. Its simulation part simulates the execution of Android applications, and explores application states. Its monitor part checks for energy inefficiency problems according to two types of misuse patterns, i.e., missing sensor listener or wake lock deactivation, and sensor data underutilization.

Figure 2 shows the high-level abstraction of NavyDroid. The simulation part is named *application execution engine*. The monitor part consists of two components, a *misuse checker* and a *sensor data utilization analyzer*. These two components check for energy bugs corresponding to the two energy inefficiency patterns, respectively.

NavyDroid takes as input an Android application's binary code and configuration files. It retrieves application components and GUI layouts from configuration files before executing the application. The *application execution engine* executes an application, and systematically explores its application states. The *misuse checker* monitors the operations on sensor listeners and wake locks during the execution. The *sensor data utlization analyzer* feeds sensor data to the application when related sensor listeners are registered. It then tracks where the sensor data propagate as the application executes, and analyzes how sensor data are utilized at different application states. At the end of the execution, NavyDroid compares sensor data utilization across explored application states to find underutilized states. It also reports misuses of sensor listeners and wake locks as output. We elaborate on these functional modules of NavyDroid in the following.

3.2 Supporting Techniques

Java Path Finder (JPF) is a testing and verification framework for Java programs. As shown in Figure 2, JPF is the base component and the supporting framework of NavyDroid. The core of JPF is a virtual machine for Java bytecode. In NavyDroid, the application execution engine executes an application in JPF's virtual machine. The implementation of NavyDroid utilizes JPF's facilities, including *instruction listener* and *native peer*.

• **Instruction listener.** Instruction listeners provide a way to inspect the execution of instructions [8]. NavyDroid keeps track of an Android application's execution by implementing instruction listeners. The *sensor data utilization analyzer* depends on this mechanism to trace the propagation of sensor data.

• **Native peer.** A *native peer class* is like a mock class. It models a JVM's native class executed by JPF's virtual machine [9]. By native peers, we simulate the Android framework APIs in order to support the normal operation of applications. Also, we model some critical classes such as `LocationListener` and `WakeLock`. In this way, we generate some analysis-related data and import them into applications.

Since Android applications frequently interact with users, their executions are often triggered by user events. This event-driven feature separates program code into different event handlers, with implicit calling relationships. However, JPF is designed for conventional Java programs and is unable to analyze event-driven programs directly. Therefore, we generate user events, and guide JPF to schedule event handlers in the simulation part.

3.3 Application Execution Engine

As the simulation part of NavyDroid, the application execution engine simulates the execution of an Android application. It enables JPF to analyze event-driven Android applications. The engine first simulates user interactions by generating sequences of user events. This process is called *event sequence generation*. The engine explores application states by systematically generating event sequences. With event sequences generated, the engine then schedules the corresponding event handlers. The effect of *evenet handler scheduling* depends on the quality of the *application execution model* (AEM). In the following sections, we introduce these processes, i.e., event sequence generation, state exploration, and event handler scheduling.

3.3.1 *Event Sequence Generation*

The first process of the application execution engine is event sequence generation. NavyDroid generates user interactions and system messages to explore application states. We first define the concept of event sequence.

Definition 1 (event sequence)**.** An event sequence *seq* is an ordered list:

$$seq = [e_1, e_2, \ldots, e_n] \quad (e_i \in E, i = 1, 2, \ldots, n),$$

where E denotes the set of events. An event sequece is a sequence of user and system events, which an application takes as input during its execution.

NavyDroid constructs the event set by static analysis. It retrieves GUI layouts of activites from the configuration files of an application, and extracts GUI widgets declared in the layout files. Each GUI widget (e.g., a button) receives a set of user actions (e.g., button clicks). We define the *candidate event set* of an activity as the union set of these user actions. The candidate event set also contains some special events, such as physical keys (i.e., *back*, *menu*, and *home*) and the *kill* system event.

The application execution engine generates events in an iterating manner. An application first launches from its entry activity and enters the *initial state*. At this point, the first event can be generated and fed to the application. At runtime, the application execution engine watches the foreground activity. When the activity waits for user interactions, the engine generates an event selected from the candidate event set of this activity. This event generation process continues until all acitivities finish, or the number

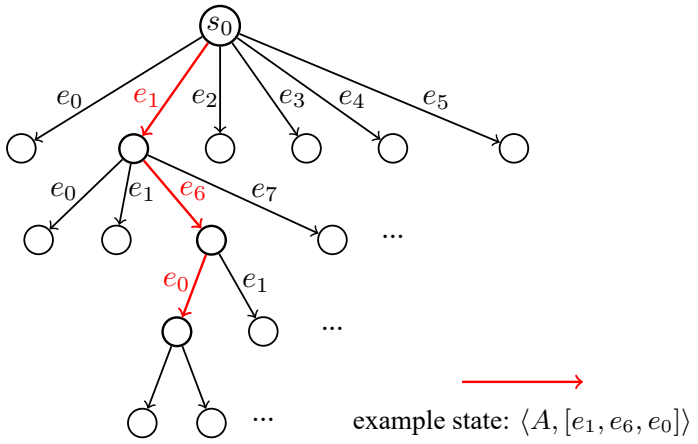


Figure 3 State tree: the model of application states and event sequences

of generated events reaches an upper bound. [1)] In the latter case, the engine generates additional *back* events to finish the running activities. When receiving a back event, the current activity is destroyed, and the previous activity comes to foreground. The application finishes when all the activities are destroyed. Therefore, by adding back events, the execution of the application terminates finally.

3.3.2 *State Exploration*

In this section, we elaborate on our approach to systematically exploring application states in parallel. The application execution engine generates event sequences in order to explore application states. We first show that we can explore application states as long as we generate all possible event sequences. We define the correspondence between application states and event sequences as follows.

Definition 2. Given an application A (with *initial state* s_0) and an input event sequence $seq = [e_0, e_1, \ldots, e_n]$, the execution of the application is in an iterating manner. In the k-th iteration ($1 \leqslant k \leqslant n$), an event e_k is fed to the application, and the application transits from state s_{k-1} to state s_k. Each event e_k should be in the *candidate event set* of state s_{k-1}. Finally, application A reaches state s_n with event sequence seq, denoted as $s_n = \langle A, seq\rangle$.

As Definition 2 shows, we can explore application states as long as we generate all possible event sequences. In order to explore application states, we generate all possible event sequences, and execute the application according to the event sequences.

In order to facilitate understanding, we model all the event sequences and application states as a rooted tree called *state tree*, as shown in Figure 3. In this tree, a node represents an application state, and an edge represents an event. The root of the tree is the initial state of the application. All the leaving edges of an non-leaf node represent the candidate events of this state. For any state in the tree, its corresponding event sequence is the path from the root node to it. As mentioned before, as event sequences have an upper bound, the depth of the tree limited to a constant *bound*.

The application execution engine can generate event sequences randomly, or in a systematic manner. JPF can be used as a model checker to exhaustively explore all possible program states, like traversing the state tree. However, when we simulate the Android framework APIs in JPF, its model checker becomes fragile and often crashes. Therefore, E-GreenDroid generates event sequences randomly for state exploration. Although it is possible to reach any application state by randomly generating event sequences, many of these event sequences are duplicated. NavyDroid takes a different approach from the random mode. It explores application states systematically to avoid event sequence duplication and improve analysis efficiency.

1) We restrict the length of event sequences in order to ensure the state exploration to finish in finite time. With a large enough bound, all representative states and event handlers can be explored, and this is sufficient for detecting energy problems.

The event sequences of an application cannot be generated statically, because the candidate event set of state s is unknown until an execution reaches s. Therefore, we design an algorithm for dynamic event sequence generation. The basic process of this algorithm is the breadth-first search (BFS) on the state tree, as is shown in Algorithm 1. We design the exploration algorithm in this BFS way for ease of parallelization. As a comparison, the depth-first search (DFS) algorithm depends on a stack, and it is error-prone for multiple processors to operate on a stack simultaneously.

Algorithm 1 State exploration

Input: Android application A;

```
 1: Q ← ∅ {empty queue}
 2: C_0 ← candidate event set of the initial state s_0
 3: for all e_0 ∈ C_0 do
 4:     seq_0 ← [e_0]
 5:     put seq_0 into Q
 6: end for
 7: while Q is not empty do
 8:     seq ← take an element from Q
 9:     execute the application with seq, and reach state s_k = ⟨A, seq⟩
10:     if length(seq) < bound then
11:         C ← candidate event set of s_k
12:         for all e_t ∈ C do
13:             seq' ← seq + [e_t] {append e_t to the end of seq}
14:             put seq' into Q
15:         end for
16:     end if
17: end while
```

We maintain a queue Q (first in first out) of event sequences (Line 1). First we put event sequences that contain exactly one *initial event* (an event which is accepted by the initial state) into Q (Lines 3–6). In each iteration, we take an event sequence seq, execute the application according to this event sequence, and reach a new state s_k (Lines 8–9). Then we append each event in the candidate event set of s_k to the end of seq to generate new event sequences (Lines 12–15).

Although the lengths of event sequences are bounded, the total number of event sequences can still be large. Executing all the event sequences will cost a huge amount of time. Therefore, we parallelize our state exploration algorithm to accelerate the exploration. As our algorithm is a kind of breadth-first search and depends on a queue, we create multiple processors and perform putting/taking operations on the queue concurrently. There are $n+1$ processors, including one master processor and n worker processors. The master processor puts initial event sequences into Q, and starts all the worker processors. Each worker processor works the same as in Algorithm 1. It takes an event sequence seq from Q, executes it, and puts newly generated event sequences into Q. Finally, the master processor stops all the worker processors when Q becomes empty.

3.3.3 *Event Handler Scheduling*

Event handler scheduling is the process that invokes event handlers corresponding to the generated event sequences. The rules of scheduling is defined by the application execution model (AEM) module. The AEM is the key to scheduling event handlers properly.

The application execution model specifies the rules of scheduling event handlers. Generally, an application execution model can be considered as an independent module. It takes an event as input, updates the application state, and determines which event handlers to invoke. The application's lifecycle state is stored in the AEM. Like event generation, the application execution engine also schedules event handlers in an iterating manner. Each time an event is fed to the application, the engine consults the AEM about the event handlers to schedule, and invokes a sequence of event handlers orderly.

The application execution model has various forms in existing approaches. GreenDroid defines AEM in the form of temporal rules, denoted as $[\psi], [\phi] \Rightarrow \lambda$. This formula represents the expected event handers

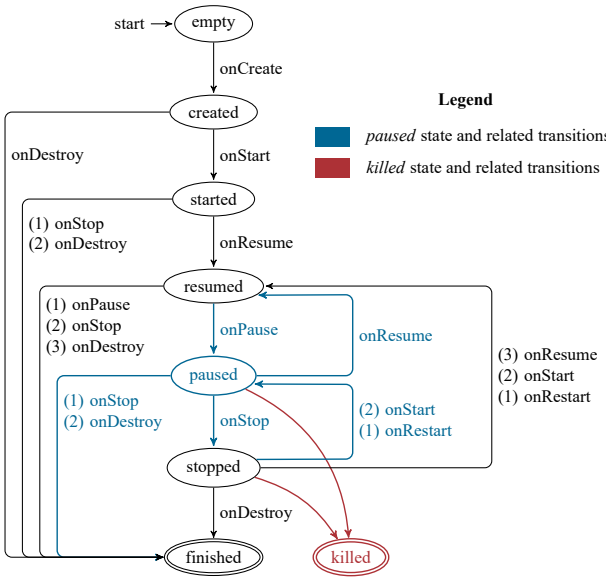


Figure 4 NavyDroid AEM model (the activity part)

λ given the execution history ψ and the current situation ϕ [23]. E-GreenDroid translates temporal rules to an abstract state machine that is practical in programming [32]. However, the AEM of E-GreenDroid is not accurate enough in some lifecycle states of Android components. Therefore, we propose a new application execution model to simulate the runtime behavior of Android applications. We present the formal definition and the improvement of our AEM as follows.

The AEM is represented as a strengthened deterministic finite automaton (DFA). The inputs and inner states of the DFA represent user events, and an application's lifecycle states, respectively. It is also enhanced with additional outputs of event handlers.

Definition 3 (application execution model)**.** An application execution model AEM is a 7-tuple, $(S, E, H, \delta, f, s_0, F)$, consisting of

- a finite set of lifecycle states (S)
- a finite set of input events (E)
- a finite set of output event handler sequences (H)
- a transition function $(\delta : S \times E \to S)$
- a scheduling function $(f : S \times E \to H)$
- a start lifecycle state $(s_0 \in S)$
- a set of final lifecycle states $(F \subseteq S)$

Our AEM differs from a common DFA in a scheduling function f and an output set H. If an application receives user event e at lifecycle state s, a list of event handlers $h = f(s, e) \in H$ is expected to schedule. Figure 4 shows the transition diagram of our application execution model. In this figure, a node represents a lifecycle state $s \in S$, and an edge represents a state transition. Scheduled event handlers are marked on the edges, correponding to the state transitions. The states and transitions of AEM associate closely with the activity lifecycle. For example, when the AEM enters the *resumed* state, the `onResume()` callback is scheduled, and the activity comes to foreground; when the AEM enters the *stopped* state, the `onPause()` and `onStop()` callbacks are scheduled, and the activity switches to background.

Compared with E-GreenDroid's state machine model, our AEM is more accurate in modeling (1) an activity that stays paused, and (2) an activity killed by the system.

An activity has an intermediate *paused* state between its *resumed* and *stopped* states. The *paused* state of an activity means that the activity is not at foreground, but still visible. An activity stays paused

Table 1 The correlation between activity state and the likelihood of the system's killing the process

Activity state	Likelihood of being killed
created, started, resumed	least
paused	more
stopped, destroyed	most

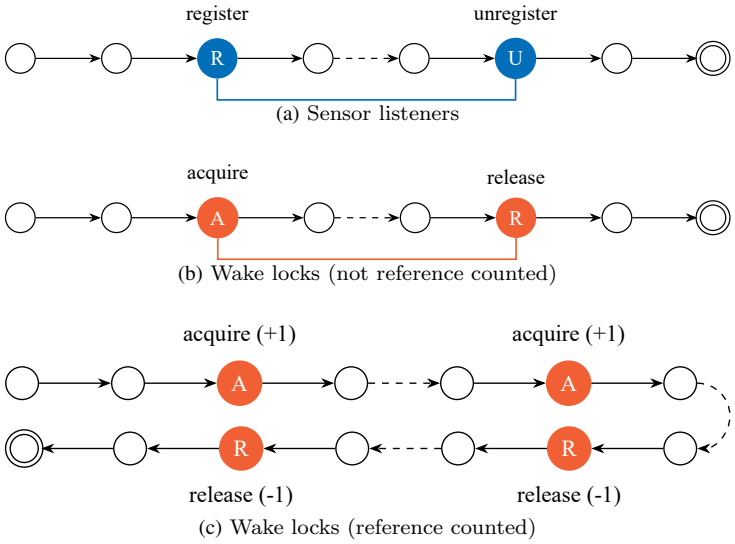


Figure 5 Use modes of sensor listeners and wake locks

when a new, translucent activity (such as a dialog) keeps open. The Android system may kill an activity in the absence of memory. Table 1 illustrates that an activity is likely to be killed at the paused, stopped, and destroyed states [1].

In summary, we construct an application execution model that specifies the event handler scheduling rules accurately. With event sequence generation, state exploration and event handler scheduling, the application execution engine simulates the runtime behavior of an application, and reaches different application states.

3.4 Sensor and Wake Lock Misuse Check

As mentioned before, the *misuse checker* is one of the components of the monitor part. It corresponds to the first energy inefficiency pattern, i.e., missing sensor listener or wake lock deactivation. The misuse checker monitors the register/unregister of sensor listeners and acquire/release of wake locks during the simulation execution of an application.

Sensor listeners and wake locks are both interfaces that request system resources. A sensor listener requests corresponding sensors to work and fetches sensor data, and a wake lock requests the CPU to stay awake. Therefore, the misuse patterns of them are similar to some extent. In the following, we discuss our misuse patterns of sensor listeners and wake locks, and the detecting methods in NavyDroid.

Sensor listener misuse patterns Figure 5(a) illustrates the usage mode of sensor listeners [24]. A sensor listener is registered for fetching sensor data. When the sensor data is no longer needed, the sensor listener should be unregistered. NavyDroid monitors and records all operations on sensor listeners. It checks execution paths for unregistered sensor listeners.

Wake lock misuse patterns When without reference counts, wake locks has behavior similar to sensor listeners. However, the usage mode and misuse pattern of reference-counted wake locks are more complicated than sensor listeners. The reference counts of wake locks are like semaphores. An acquire

Table 2 Common patterns of wake lock misuses and existing tools' capability

Misuse pattern	Number of issues	Related to energy waste?	E-GreenDroid solvable?	NavyDroid solvable?
Unnecessary wakeup	11	Yes	-	-
Wake lock leakage	10	Yes	Solvable	Solvable
Permature lock releasing	9	No	-	-
Multiple lock acquisition	8	Yes	-	Solvable
Inappropriate lock type	8	Yes	-	-
Problematic timeout setting	3	No	-	-
Inappropriate flags	2	Yes	-	-
Permission errors	2	No	-	-

operation increases the counter, while a release operation decreases the counter. A wake lock keeps held as long as its reference count is larger than zero. Therefore, in an execution path, the number of release operations of a wake lock should be no less than the number of acquire operations. Figure 5(b) and Figure 5(c) illustrates the two different usage modes of wake locks.

An empirical study shows the eight common misuse patterns of wake locks [25]. As Table 2 shows, five out of these patterns relate to energy inefficiency. The *wake lock leakage* pattern corresponds to wake locks without reference counts, and the *multiple lock acquisition* pattern corresponds to wake locks with reference counts. Existing work, such as GreenDroid and E-GreenDroid, only detects the former pattern, while NavyDroid addresses these two patterns of wake lock misuses. We do not address all misuse patterns related to energy inefficiency, because some patterns lack detection criteria.

In summary, the misuse checker of NavyDroid monitors the operations of sensor listeners and wake locks to check for misuses. It supports a new misuse pattern of wake locks. In the analysis report, NavyDroid records the misused sensor listeners and wake locks with related application paths.

3.5 Sensor Data Utilization Analysis

Sensor data utilization analyzer is another component of the monitor part. It corresponds to the second energy inefficiency pattern, i.e., sensor data underutilization. The sensor data utilization analyzer generates and propagates sensor data, and evaluates how sensor data is utilized at different application states.

As mentioned before, the sensor itself consumes energy to retrieve sensor data, thus applications should use sensor data efficiently. The utilization of sensor data can vary among application states. The analyzer reports application states with low utilization of sensor data. We do not analyze wake lock utilization, because wake locks only relate to CPU power management, and do not retrieve or use data like sensors.

The analysis of sensor data utilization can be divided into the following three phases.

Tainting. The analyzer generates sensor data object, each with a unique taint mark. The sensor data are generated from a data pool. NavyDroid feeds sensor data to the application when the application registers a sensor listener. When sensor data objects are constructed, NavyDroid assigns taint marks to these objects.

Propagation. The analyzer keeps track of the transformation of sensor data, and propagates taint marks via data flow. The propagation is at the level of bytecode instruction. The result of an instruction is tainted if one of the operands of this instruction is tainted. NavyDroid follows a collection of propagation rules [23]. The tainting and propagating phase both leverages JPF's *object attribution* functionality.

Evaluation. The analyzer evaluates the utilization of sensor data at executed application states. We calculate the *data utilization coefficient* (DUC) of each application state as [21]:

$$\mathrm{DUC}(d,s) = \frac{\mathrm{usage}(s,d)}{\max_{s' \in S, d' \in D} \mathrm{usage}(s',d')} \tag{1}$$

The DUC value of sensor data d at state s is defined as the ratio between the usage of d at state s and the maximum usage of any sensor data at any state. The usage of sensor data d at state s is defined as [21]:

$$\text{usage}(s, d) = \sum_{i \in \text{instr}(s,d)} \text{weight}(i, s) \times \text{rel}(i) \tag{2}$$

In Equation 2, $\text{instr}(s, d)$ denotes the set of bytecode instructions executed after d is fed to the application. Indicator function $\text{rel}(i)$ tests whether an instruction i uses any data with the same mark as d has. Function $\text{weight}(i, s)$ assigns a weight to instruction i to measure the benefits that i brings to the user.

After calculating DUC values of application states, we filter out application states with low DUC values. A low DUC value represents a low utilization of sensor data, and indicates an energy inefficiency bug. In this way, the analyzer identifies the inefficiency in sensor data utilization at certain application states. In the analysis report, NavyDroid records low sensor data utilization with related application paths and states.

4 Evaluation

In this section, we demonstrate the effectiveness and efficiency of our approach by experiments. NavyDroid is extended from E-GreenDroid, as E-GreenDroid is a state-of-the-art energy inefficiency detection tool. We replace the application execution model of E-GreenDroid with our strenthened DFA, in order to simulate the paused and killed states of activities accurately. First, we compare the effectiveness of NavyDroid and E-GreenDroid. On one hand, we evaluate whether NavyDroid can detect the equivalent energy inefficiency bugs that E-GreenDroid reports. On the other hand, we evaluate the enhanced abilities of NavyDroid as compared with E-GreenDroid. Second, we compare the efficiency of E-GreenDroid, sequential NavyDroid, and parallel NavyDroid. We aim to answer the following three research questions:

- **RQ1 (Ability equivalence):** Does NavyDroid hold the abilities as E-GreenDroid does; i.e., can NavyDroid conduct effective analysis on those applications with energy inefficiency bugs that E-GreenDroid can effectively analyze?
- **RQ2 (Ability enhancement):** Can NavyDroid detect energy inefficiency problems that E-GreenDroid is not able to detect?
- **RQ3 (Efficiency of state exploration):** With its (parallel) state exploration algorithm, does NavyDroid analyze applications more efficiently than E-GreenDroid?

4.1 Experimental Setup

We selected different test subjects for the three research questions. For RQ1, we selected all the 13 Android applications that were used in the evaluation of E-GreenDroid [32]. The basic information of these 13 applications is shown in Table 3. E-GreenDroid reports energy inefficiency bugs in all of them. For RQ2, we selected six Android applications as test subjects. The basic information of these applications is shown in Table 4. They are all real-world applications with a relatively large scale (more than 1,000 lines of code). For RQ3, we took all the applications in RQ1 and RQ2 as test subjects and evaluated the efficiency of analyzing them. We obtained the source codes of all these applications, as they are all open source projects.

Our experiments were conducted on a quad-core computer with Intel Core i7 CPU and 8GB RAM. The machine was installed with Ubuntu 16.04 LTS. We compiled each application on Android 5.0 for our experiments. For RQ1 and RQ2, we controlled E-GreenDroid to randomly generate 5,000 event sequences, and NavyDroid to systematically explore application states. The lengths of event sequences were limited to six at most. This length is enough for E-GreenDroid and NavyDroid to explore considerable application states and detect energy inefficiency bugs. For RQ3, we ran parallel state exploration with four working threads.

Table 3 Information and analysis results of RQ1's test subjects

Application	Revision	Lines of code	E-GreenDroid results[1]	NavyDroid results[1]	Equivalent[2]
AndTweet	V-0.2.4	8,908	WLM	WLM	Yes
AAT	V-0.9-alpha	52,800	SDU	SDU	Yes
BabbleSink	R-d12879a	1,718	WLM	WLM	Yes
CWAC-Wakeful	R-d984b89	896	WLM	WLM	Yes
GPSLogger	R-15	659	SLM, SDU	SLM, SDU	Yes
GPSLogger-new	-	789	SLM, SDU	SLM, SDU	Yes
LocWriter2	V-0.1.1	1,542	SDU	SDU	Yes
OmniDroid	R-863	12,427	SDU	SDU	Yes
OsmDroid	R-750	18,091	SDU	SDU	Yes
Recycle Locator	R-68	3,241	SLM	SLM	Yes
RedBlackTree	R-0	483	WLM	WLM	Yes
Sofia Public Transport Nav.	R-114	1,443	SDU	SDU	Yes
Ushahidi	R-9d0aa75	10,186	SLM	SLM	Yes

[1] We denote **SLM** as *sensor listener misuse*, **WLM** as *wake lock misuse*, and **SDU** as *sensor data underutilization*.

[2] Whether E-GreenDroid and NavyDroid report the equivalent results.

We ran both E-GreenDroid and NavyDroid to diagnose all the test subjects, and examined the analysis reports to compare their abilities of locating energy inefficiency bugs. For RQ3, we evaluated and compared the efficiency of (1) E-GreenDroid exploring application states randomly; (2) NavyDroid exploring application states sequentially (with a single thread) and (3) NavyDroid exploring application states parallely (with multiple threads). Also, we showed by examples that the systematic state exploration algorithm of NavyDroid improves efficiency by eliminating duplicated event sequences. In the following, we elaborate on our experimental results with respect to the three research questions.

4.2 RQ1: Ability Equivalence

In order to answer RQ1 about the ability equivalence of NavyDroid and E-GreenDroid, we ran both E-GreenDroid and NavyDroid to analyze the test subjects. There are three types of energy inefficiency bugs, namely, *sensor listener misuse*, *wake lock misuse*, and *sensor data underutilization*. NavyDroid and E-GreenDroid detected the same types of energy inefficiency bugs.

Table 3 lists the analysis results collected from E-GreenDroid and NavyDroid. E-GreenDroid and NavyDroid reported all misuses of sensor listeners and wake locks if they detect these two types of bugs. For sensor data underutilization bugs, they reported all suspicious application states with DUC values less than 1.0 for user's decision. In our experiments, we defined a DUC value that is less than 0.5 as *severe* underutilization. We considered a severe sensor data underutilization at an application state as an energy inefficiency bug.

We compared the detailed information in the analysis reports of E-GreenDroid and NavyDroid to further demonstrate the equivalence of energy inefficiency bugs. For energy inefficiency bugs of types *sensor listener misuse* and *wake lock misuse*, we compared the execution traces of them. Only two bugs with the same type and the equivalent execution traces are considered as equivalent. [2)] For energy inefficiency bugs of type *sensor data underutilization*, we considered all application states with DUC values less than 1.0, and compared the distributions of DUC values. As is shown in Figure 6(a), 6(b), 6(c)

2) The execution traces in analysis reports are not exactly the same because NavyDroid represents application states in an alternative way. We define two execution traces as equivalent when their event handlers and components are the same.

Table 4 Information and analysis results of RQ2's test subjects

Application	Revision	Lines of code	E-GreenDroid results[1]	NavyDroid results[1]	Improved[2]	Cause[3]
AndroidRun	R-dbf6428	1,649	SLM[4]	SLM (1) [5]	Yes	AEM
VLC	R-abe60f5	6,839	NPD	WLM (1)	Yes	MLA
CSipSimple[6]	R-152	13,107	NPD	NPD (1)	No	-
TomaHawk	R-543c3b9	4,601	NPD	WLM (2)	Yes	AEM, MLA
AndrOBD	R-d1a6ba0	24,161	NPD	WLM (2)	Yes	AEM
MTPMS	R-935ceeb	1,217	NPD	SLM (1)	Yes	MLA

[1] We denote **NPD** as *no problem detected*, **SLM** as *sensor listener misuse*, and **WLM** as *wake lock misuse*. The numbers in parentheses indicate the number of bugs that NavyDroid detected for each application.

[2] Explaining whether the results of NavyDroid are better than E-GreenDroid.

[3] Explaining why NavyDroid reports better analyzing results than E-GreenDroid. **AEM** represents the improvement in application execution model, and **MLA** represents the ability of monitoring the misuse pattern of multiple lock acquisition.

[4] The SLM bug that E-GreenDroid reports in AndroidRun is a false positive. We will discuss this in detail.

[5] NavyDroid reports multiple energy inefficiency problems, but they are caused by one defect after manual analysis. We exclude the essentially equivalent reported bugs.

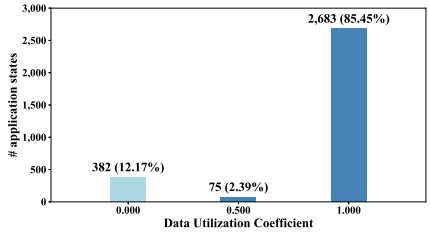
[6] CSipSimple has six problematic revisions according to the empirical study. We compress these six revisions because the results are all NPDs.

and 6(d), the DUC value distribution for GPSLogger and OsmDroid in the report of E-GreenDroid and NavyDroid were essentially the same. In summary, we can answer RQ1 that NavyDroid locates all the energy inefficiency bugs reported by E-GreenDroid and holds the abilities of E-GreenDroid.

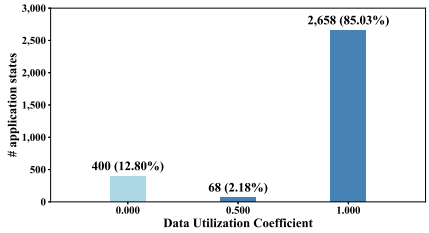
4.3 RQ2: Ability Enhancement

In order to answer RQ2 about the ability enhancement of NavyDroid as compared with E-GreenDroid, we analyzed several new test subjects with E-GreenDroid and NavyDroid. We selected six real-world Android applications as test subjects. Table 4 lists the basic information and analysis results of these applications. We collected and compared the analysis reports of E-GreenDroid and NavyDroid in the same way as RQ1. E-GreenDroid either detected no energy inefficiency bugs in an application, or reported false positives. Whereas, NavyDroid reported seven energy inefficiency bugs in six applications. We discuss some test subjects and the detected problems as follows.

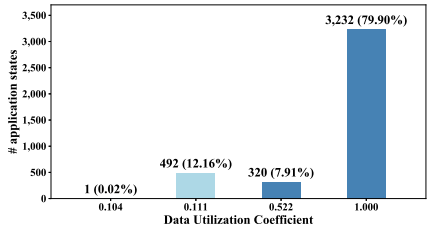
AndroidRun AndroidRun is a running and biking aiding application [3]. It tracks the location and calculates the speed of users. NavyDroid reported a sensor listener misuse in AndroidRun. The main activity registers a location listener (a kind of sensor listeners) in its `onCreate()` callback to collect GPS data. The unregister operation of this location listener is in the `onDestroy()` callback. However, if the activity gets killed by the Android system when it switches to the background, the `onDestroy()` callback will never be invoked, leaving the location listener still registered. E-GreenDroid also reported this sensor listener misuse. However, in E-GreenDroid's monitor part, after all the events in a event sequence are fed into the application and the corresponding event handlers are scheduled, the misuse checker immediately checks for sensor listener misuses. By this time, activities have not finished their lifecycles yet. Therefore, we considered the reported sensor listener misuse as a false positive. NavyDroid avoids this false positive, but it can still report the misuse of this location listener, because it properly simulates the killed state in the application execution model.



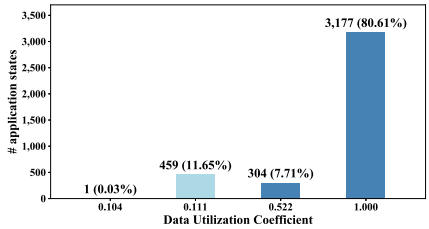
(a) DUC values of GPSLogger analyzed by E-GreenDroid



(b) DUC values of GPSLogger analyzed by Navy-Droid



(c) DUC values of OsmDroid analyzed by E-GreenDroid



(d) DUC values of OsmDroid analyzed by Navy-Droid

Figure 6 DUC value distribution bar charts

VLC VLC is an open source media player and framework [14]. NavyDroid reported an unbalanced acquires and releases of a wake lock in VLC. The `VideoPlayerActivity` plays user-specified videos. It acquires a wake lock to keep the screen on while playing the video. When a user plays or pauses video playing, the wake lock gets acquired or released. However, the wake lock can also be acquired when this activity is resumed and finishes loading the media file. If a user navigates to this activity and clicks the play/pause button twice, the wake lock will be acquired twice but released only once. According to the multiple lock acquisition pattern, a wake lock misuse happens, and NavyDroid detected this misuse. E-GreenDroid failed to detect this wake lock misuse because it does not consider the number of acquires and releases of wake locks.

CSipSimple CSipSimple is a communication application, which supports SIP (Session Initiation Protocol) connection over the Internet [6]. A recent empirical study [25] reports wake lock misuses in CSipSimple. However, both E-GreenDroid and NavyDroid detected no energy inefficiency problems in it. CSipSimple maintains a `SipService` at background to register accounts. When there is successful registeration of an account, the service acquires a wake lock to keep the screen on. The number of acquire operations is not restricted, as the registeration of accounts can be repeated. Nevertheless, the service sets the wake lock to be non reference-counted, thus the wake lock will not keep held after one release operation. So we considered that CSipSimple has no misuses about this wake lock, and NavyDroid reported a reasonable analysis result.

TomaHawk TomaHawk is a music player which supports multi-source media [13]. NavyDroid reported two wake lock misuses in TomaHawk. As described in the motivating example in Section 2.2, the two bugs occur in different cases. If a user clicks the play/pause button twice after the media player gets prepared, the wake lock is acquired twice but released once, and remains being held as its reference count is larger than zero. E-GreenDroid fails to detect this wake lock misuse because it does not monitor the reference counts of wake locks. Another bug occurs when the Android system kills the activity and the service associated with media playing, in which case the `onDestroy()` callback is not invoked, causing wake lock leakage. E-GreenDroid fails to detect this wake lock misuse because it does not simulate the killed state of an activity.

Table 5 Average execution times of sequential and parallel state exploration algorithms

Application	Number of explored event sequences	Average execution time (E[1]) /s	Average execution time (N1[1]) /s	Average execution time (N4[1]) /s
AAT	1,456	1.284	2.048 (+60%[2])	0.816 (−60%[3]) (−36%[4])
AndrOBD	1,645	1.944	3.376 (+74%)	1.122 (−67%) (−42%)
AndroidRun	189	1.710	3.101 (+81%)	1.413 (−54%) (−17%)
AndTweet	1,456	1.313	1.886 (+44%)	0.801 (−58%) (−39%)
BabbleSink	189	1.084	1.783 (+64%)	0.693 (−61%) (−36%)
CWAC-Wakeful	189	1.108	1.693 (+66%)	0.624 (−63%) (−39%)
GPSLogger	5,000	1.310	1.954 (+49%)	0.778 (−60%) (−41%)
GPSLogger-new	5,000	1.388	2.123 (+53%)	0.854 (−60%) (−38%)
LocWriter2	1,456	1.244	1.992 (+60%)	0.786 (−61%) (−37%)
OmniDroid	5,000	3.952	6.360 (+61%)	2.485 (−61%) (−37%)
OsmDroid	5,000	1.926	3.093 (+61%)	1.331 (−57%) (−31%)
Recycle Locator	189	1.394	2.016 (+45%)	0.772 (−62%) (−45%)
RedBlackTree	5,000	1.062	1.678 (+58%)	0.616 (−63%) (−42%)
Sofia Public Transport Nav.	432	1.778	3.421 (+92%)	1.461 (−57%) (−18%)
TomaHawk	1,456	1.222	2.035 (+67%)	0.804 (−60%) (−34%)
MTPMS	189	1.078	1.720 (+60%)	0.683 (−60%) (−37%)
Ushahidi	432	1.596	2.468 (+55%)	1.012 (−59%) (−37%)
VLC	1,277	1.326	2.107 (+59%)	0.892 (−58%) (−33%)

[1] **E** denotes E-GreenDroid's random event sequence generation, **N1** denotes NavyDroid's state exploration algorithm with one single thread, and **N4** denotes NavyDroid's state exploration algorithm with four threads.
[2] Rate of change relative to E-GreenDroid's average execution time.
[3] Rate of change relative to the average execution time of single-threaded NavyDroid.
[4] Rate of change relative to E-GreenDroid's average execution time.

4.4 RQ3: Efficiency of State Exploration

In order to evaluate the efficiency of the parallel exploration algorithm, we took all the applications in RQ1 and RQ2 as test subjects. We analyzed these applications with (1) the random event sequence generation of E-GreenDroid, (2) the sequential state exploration of NavyDroid and (3) the parallel state exploration of NavyDroid. For all three cases, we compare the average execution time of applications. As discussed in experimental setup, we ran parallel exploration with four working threads. Also, we restrict the sequential algorithm to work on one CPU core. We controlled E-GreenDroid to generate 5,000 event sequences for each test subject. For NavyDroid, because some of the test subjects have too many states to explore, we restricted the state exploration to generate at most 5,000 event sequences.

We recorded the number of executed event sequences, and the execution time for the three cases. Then we calculated the average time of each test subject. Table 5 and Figure 7 presents the analysis results of the three cases. Although the execution time of NavyDroid is larger than that of E-GreenDroid by about 50%, the parallelization of the state exploration algorithm reduces execution time by about 60%. Overall, NavyDroid saves about 40% execution time compared with E-GreenDroid.

Moreover, NavyDroid improves efficiency by generating more unrepeated event sequences. The exploration algorithm ensures that all the generated event sequences are unrepeated. However, if generated randomly, the event sequences will contain repeated ones. We selected AndTweet and OmniDroid as representatives in order to show the efficiency of removing duplicated event sequences. Our state exploration

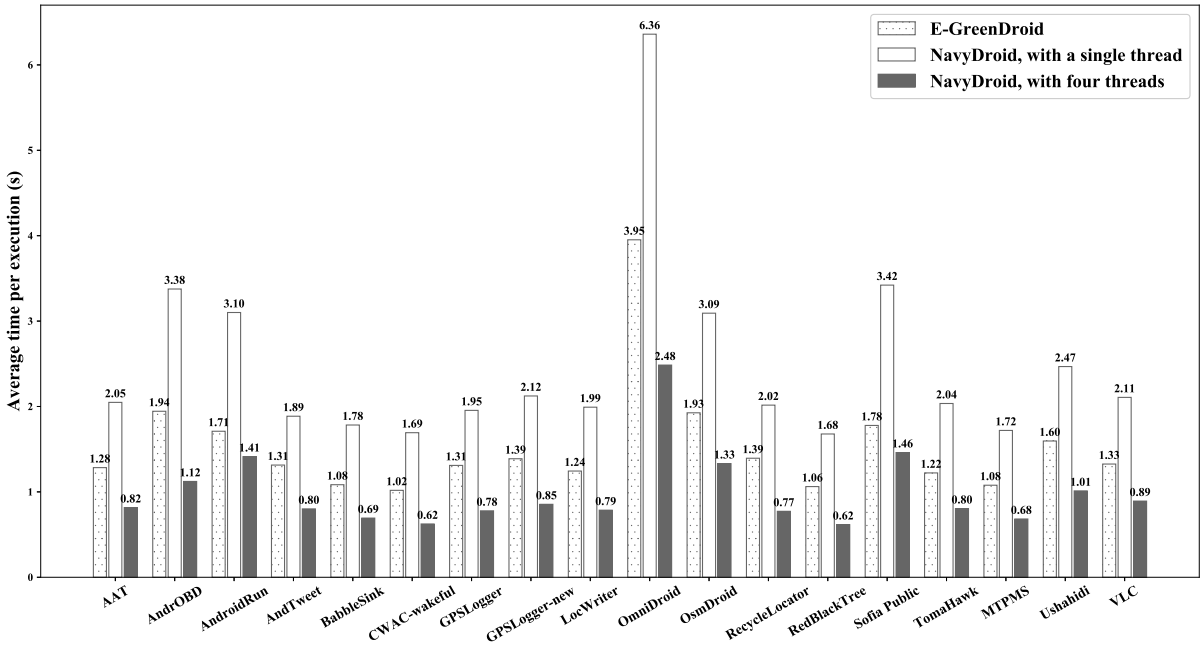


Figure 7 Average execution time of sequential and parallel state exploration

algorithm generates 1,456 and 13,384 event sequences for AndTweet and OmniDroid, respectively.

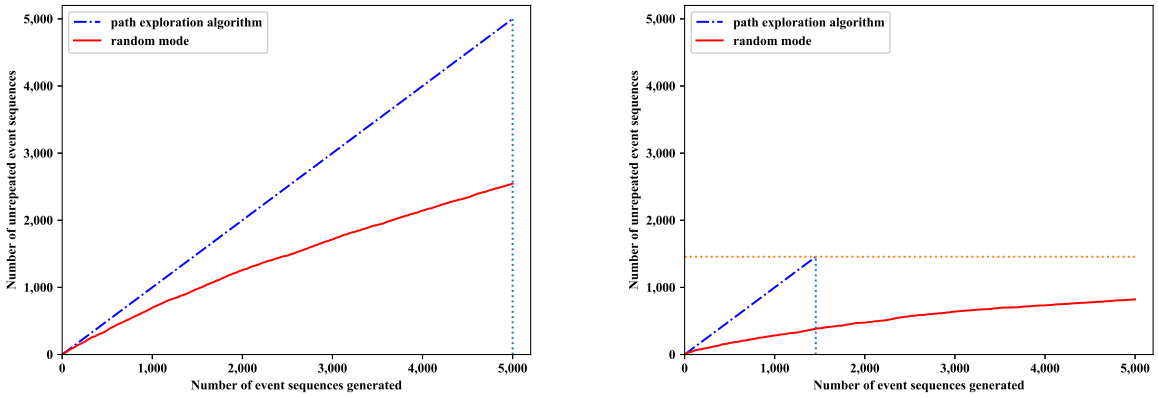
Figure 8 shows the trend of unrepeated event sequence number as the total event sequence number increases. In the figure, the faster the curve rises, the higher the efficiency of state traversal is. Our state exploration algorithm reaches the optimal efficiency of state traversal (every generated event sequence is unrepeated). As shown in Figure 8(a), for OmniDroid, in 5,000 executions, the random mode only generates 2,546 valid (unrepeated) event sequences. The efficiency is only 51% of the state exploration algorithm. As shown in Figure 8(b), for AndTweet, as the total number of unrepeated event sequences is 1,456, the state exploration algorithm stops after generating 1,456 event sequences. However, the random mode only generates 384 unrepeated event sequences in 1,456 executions, and 822 valid event sequences in 5,000 executions.

Therefore, it is not efficient to generate event sequences randomly due to lots of repeated event sequences. Also, as the number of randomly generated event sequences increases, the number of repeated ones grows and the efficiency decreases. Our state exploration algorithm exceeds random generation in efficiency by removing duplicated event sequences.

In summary, the state exploration algorithm of NavyDroid is efficient in two aspects. First, the state exploration algorithm generates more unrepeated event sequences than random generation within the same time. Second, by parallelizing the exploration algorithm, lots of time can be saved. If necessary, we can extend our parallel exploration algorithm to distributed systems and reduce execution time by running on multiple servers.

5 Discussions

Like E-GreenDroid, NavyDroid is also implemented on top of JPF. We choose JPF as the analysis framework because of its functionality of intercepting the execution of Java applications. One limitation of JPF is that it can only analyze traditional Java programs. In our approach, the application execution model guides the simulation execution of Android applications. It stores an application's lifecycle state, and determines the scheduling of event handlers. Our AEM may still be inaccurate when simulating the execution of Android applications in some cases. However, it is highly extensible and can be easily updated.



(a) OmniDroid (b) AndTweet

Figure 8 The trend of unrepeated event sequence number

In this paper, we present NavyDroid as an energy inefficiency diagnosis tool. Furthermore, the application execution model can serve as a general dynamic analysis tool, since it generates event sequences for applications. As discussed before, the diagnosis tool consists of a simulation part and a monitor part. The application execution model is the key to the simulation part. It defines the scheduling policies of event handlers, and provides native framework libraries for applications. We can assemble the simulation part with different monitor parts, i.e., components with different monitor points and problem patterns. With different monitor policies, NavyDroid can detect more bugs in Android applications. Therefore, the application execution model and the simulation part can become a more general dynamic analysis tool for Android applications.

Currently, NavyDroid is able to detect two wake lock misuse patterns. NavyDroid cannot address the *unnecessary wakeup* pattern, which is the most common pattern of wake lock misuse according to the empirical study [25]. We plan to extend NavyDroid to analyze this pattern in future. There are also some other misuse patterns of wake locks such as *inappropriate lock type* and *inappropriate flags*. However, these two patterns currently lack feasible diagnosis criteria, and thus are difficult to detect automatically.

The Android system hands over part of the energy management power to developers by exposing some resource management APIs to developers. Thus, misuse patterns are usually related to these resource management APIs. In this work, we select sensor listener and wake lock as two representative APIs, and derive three misuse patterns from them. We can probably find more misuse patterns from resource related APIs, so as to detect more bugs in applications. For instance, `MediaRecorder` is used to record audio and video, and `Camera` is used to take pictures, and they can potentially cause energy inefficiency [30].

6 Related Work

Our work relates to existing studies of several research topics, which includes energy inefficiency analysis and wake lock misuse detection. In this section, we discuss some representative pieces of work in recent years.

Energy efficiency analysis In recent years, researchers have proposed various energy inefficiency diagnosis approaches for smartphone applications. Some pieces of work use static analysis techniques to detect energy inefficiencies. Pathak et al. detected no-sleep energy bugs in Android applications with reaching-definition data-flow analysis [30]. Their work is probably the first to explore and define the characteristics of no-sleep energy bugs. Many researchers proposed apporaches based on dynamic analysis techniques to avoid false positives in static analysis. Liu et al. characterised energy inefficiency

caused by sensor data underutilization, and proposed GreenDroid that searches application states for low sensor data utilization [23]. GreenDroid detects energy inefficiency bugs with taint-based dynamic data-flow analysis. Zhang et al. presented a framework named ADEL to detect energy leaks of network data [34]. ADEL is similar to GreenDroid in evaluating the utilization of program data. CyanDroid systematically generates multi-dimensional sensor data to reproduce energy inefficiency bugs that require specific sensor data to manifest [20]. Wang et al. updated and optimized the execution simulation process of GreenDroid [32].

Some approaches focus on optimizing and repairing energy inefficiency problems. Ma et al. presented eDoctor, a practical tool that captures an application's time-varying behavior in order to identify the abnormal behavior of an application [27]. It suggests repair solutions to users, and helps regular users troubleshoot energy inefficiency problems. Manotas et al. presented a general framework that automatically selects the most energy-efficient library implementations for optimizing Java applications [28]. Bouquet detects and bundles HTTP requests at runtime in order to reduce the energy consumption of HTTP requests [19]. Li et al. proposed an approach to fixing sensor data underutilization automatically through instrumentation of applications [22]. Banerjee et al. developed EnergyPatch, a framework that detects, validates and repairs energy inefficiencies in Android applications [15]. EnergyPatch uses a static analysis technique to detect potential energy bugs, and validates the presence of energy bugs with a dynamic analysis technique.

Energy consumption estimation Energy consumption estimation profiles energy hotspots (i.e., components that consume the most energy) in an application. A common practice with this type of approaches is to execute the application with different test cases, while monitoring the energy consumption of the device. Pathak et al. proposed Eprof that records invocations of system APIs, and estimates the energy consumption using a power model [29]. Besides Eprof, several subsequent pieces of work proposed more fine-grained profiling techniques for estimating energy consumption. Both eLens [17] and vLens [18] estimate energy consumption for each line of source code in an application. Banerjee et al. proposed a hardware-software hybrid approach to systematically generating test inputs that lead to energy bugs and energy hotspots [16]. Energy consumption estimation approaches relate to energy inefficiency diagnosis approaches as they identify energy hotspots. However, energy hotspots are not equivalent to energy inefficiency bugs. The concept of energy inefficiency bugs emphasizes that the energy consumption is not necessary and brings no user benefits.

Wake lock misuse detection Some severe energy problems associate with misuses of wake locks. Pathak et al. detected energy bugs caused by wake lock leakage using data-flow analysis [30]. The approach proposed by Vekris et al. also verifies wake lock misuses according to a collection of policies [31]. GreenDroid and E-GreenDroid monitor and record the operations on wake locks during simulation executions [23,32]. WLCleaner is another approach that repairs wake lock issues at runtime [33]. However, these pieces of work only address energy inefficiency bugs caused by wake lock leakage. Liu et al. conducted an empirical study to understand common wake lock usage in practice, and summarized eight common patterns of wake lock misuses [25]. Our work detects multiple wake lock misuses, including wake lock leakage, and unbalanced acquires and releases.

7 Conclusion

In this article, we have proposed an approach to detecting energy inefficiency problems in Android applications. Our approach executes Android applications by generating event sequences and scheduling event handlers. The event handler scheduling is guided by an application execution model derived from Android specifications. We also designed a parallel algorithm to explore application states systematically. During the execution, it monitors the operations on sensor listeners and wake locks to detect misuses. It also evaluates sensor data utilization using a taint-based dynamic analysis.

Our approach exceeds existing approaches by a more accurate application execution model, and more energy inefficiency patterns. We implemented our approach as a prototype tool named NavyDroid, and evaluated it with real-world applications. The experimental results demonstrate its effectiveness and efficiency in detecting energy inefficiency problems.

In future, we plan to further extend this work to support more patterns of energy inefficiency problems. We also plan to re-implement our tool on top of the Android virtual machine. At present, dynamic analysis tools can only simulate the execution of Android applications and may be inconsistent with the actual execution of applications. The analysis can be more accurate by inspecting real application executions inside the Android virtual machine.

Acknowledgements This work was supported in part by National Key R&D Program (Grant #2017YFB1001801), National Natural Science Foundation (Grants #61690204, #61472174) of China. 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 Android activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html, 2017.
2 Android application fundamentals. https://developer.android.com/guide/components/fundamentals.html, 2017.
3 Android run. https://sourceforge.net/projects/androidrun/, 2017.
4 Android location strategies. https://developer.android.com/guide/topics/sensors/sensors_overview.html, 2017.
5 Android sensors usage. https://developer.android.com/guide/topics/sensors/sensors_overview.html, 2017.
6 Csipsimple. https://github.com/r3gis3r/CSipSimple, 2017.
7 Google Play. https://en.wikipedia.org/wiki/Google_Play, 2017.
8 Java pathfinder listeners wiki page. https://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/listener, 2017.
9 Java pathfinder mji wiki page. https://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/mji, 2017.
10 Java pathfinder wiki page. https://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/what_is_jpf, 2017.
11 Managing android device awake state. https://developer.android.com/training/scheduling/wakelock.html, 2017.
12 Powermanager.wakelock class. https://developer.android.com/reference/android/os/PowerManager.WakeLock.html, 2017.
13 Tomahawk. https://github.com/tomahawk-player/tomahawk-android, 2017.
14 Vlc. https://github.com/mstorsjo/vlc-android, 2017.
15 Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*, 2017.
16 Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 588–598. ACM, 2014.
17 Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101. IEEE, 2013.
18 Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 78–89. ACM, 2013.
19 Ding Li, Yingjun Lyu, Jiaping Gui, and William G. J. Halfond. Automated Energy Optimization of HTTP Requests for Mobile Applications. In *IEEE/ACM 38th International Conference on Software Engineering*, ICSE'16, pages 249–260. IEEE, 2016.
20 Qiwei Li, Chang Xu, Yepang Liu, Chun Cao, Xiaoxing Ma, and Jian Lü. Cyandroid: Stable and effective energy inefficiency diagnosis for android apps. *Science China Information Sciences*, 60:012104, 2017.
21 Yepang Liu, Chang Xu, and Shing-Chi Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *IEEE International Conference on Pervasive Computing and Communications*, PerCom'13, pages 2–10. IEEE, 2013.
22 Yuanchun Li, Yao Guo, Junjun Kong and Xiangqun Chen. Fixing sensor-related energy bugs through automated sensing policy instrumentation. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, ISLPED'15, pages 321–326, 2015.
23 Yepang Liu, Chang Xu, Shing-Chi. Cheung, and Jian Lü. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, pages 911–940, 2014.
24 Yepang Liu. Percom 2013 Presentation. http://sccpu2.cse.ust.hk/andrewust/files/PerCom_2013_Presentation.pdf, 2013.

25 Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '16, pages 396–409. ACM, 2016.

26 Yi Liu, Jue Wang, Chang Xu and Xiaoxing Ma. NavyDroid: Detecting Energy Inefficiency Problems for Smartphone Applications. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*. ACM, 2017.

27 Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 57–70. USENIX Association, 2013.

28 Irene Manotas, Lori Pollock, and James Clause. SEEDS: A Software Engineers Energy-Optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 503–514. ACM, 2014.

29 Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 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*, EuroSys '12, pages 29–42. ACM, 2012.

30 Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 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*, MobiSys '12, pages 267–280. ACM, 2012.

31 Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12. USENIX Association, 2012.

32 Jue Wang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lü. E-greendroid: Effective energy inefficiency analysis for android applications. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware*, pages 71–80. ACM, 2016.

33 Xigui Wang, Xianfeng Li, and Wen Wen. Wlcleaner: Reducing energy waste caused by wakelock bugs at runtime. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, DASC '14, pages 429–434. IEEE, 2014.

34 Lide Zhang, Mark S. Gordon, Robert P. Dick, Z. Morley Mao, Peter Dinda, and Lei Yang. Adel: An automatic detector of energy leaks for smartphone applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 363–372. ACM, 2012.

35 Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '10, pages 105–114. ACM, 2010.