EnergyPatch: Repairing Resource Leaks to Improve Energy-efficiency of Android Apps

Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga and Abhik Roychoudhury

Abstract—Increased usage of mobile devices, such as smartphones and tablets, has led to widespread popularity and usage of mobile apps. If not carefully developed, such apps may demonstrate energy-inefficient behaviour, where one or more energy-intensive hardware components (such as Wifi, GPS, *etc*) are left in a high-power state, even when no apps are using these components. We refer to such kind of energy-inefficiencies as energy bugs. Executing an app with an energy bug causes the mobile device to exhibit poor energy consumption behaviour and a drastically shortened battery life. Since mobiles apps can have huge input domains, therefore exhaustive exploration is often impractical. We believe that there is a need for a framework that can systematically detect and fix energy bugs in mobile apps in a scalable fashion. To address this need, we have developed EnergyPatch, a framework that uses a combination of static and dynamic analysis techniques to detect, validate and repair energy bugs in Android apps. The use of a light-weight, static analysis technique enables EnergyPatch to quickly narrow down to the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially buggy program paths using a dynamic analysis technique helps in validations of the reported bugs and to generate test cases. Finally, EnergyPatch generates repair expressions to fix the validated energy bugs. Evaluation with real-life apps from repositories such as F-droid and Github, shows that EnergyPatch is scalable and can produce results in reasonable amount of time. Additionally, we observed that the repair expressions generated by EnergyPatch could bring down the energy consumption on tested apps up to 60%.

Index Terms—Mobile Apps; energy bugs; non-functional testing; energy-aware test generation

1 Introduction

Over the recent years there has been an increased usage of complex applications on battery powered mobile devices, such as smartphones and tablets. Such mobile applications or apps exploit a wide variety of sensors and other hardware components available on modern smartphones to provide a diverse set of functionalities. There is however one factor that greatly limits the usage of such apps. Battery power on mobile devices is often a constrained resource. Therefore, it is worthwhile to test and remove energy-inefficiencies in mobile apps before deployment.

In this paper, we shall present our framework (and tool) EnergyPatch, that can help app-developers to detect, validate and repair a specific class of energy-inefficiencies in mobile apps, which we refer to as energy bugs. Executing an application containing an energy bug may cause the mobile device to consume excessive amounts of battery power even after the buggy application has completed execution and there is no user-activity. Such excessive energy consumption can drastically reduce the battery life of a mobile device in a relatively short period of time[1]. In our recent work [1], we observed that inappropriate usage of energy-intensive, hardware components (such as Wifi, GPS) or power management utilities (such as Android Wakelocks) may give rise to energy bugs. We also observed that such hardware components/power management utilities can only be accessed by an app through a predefined set of system call APIs. These observations indicate that inappropriate usage of system call APIs that provides access to such hardware resources/power management utilities leads to energy bugs, resulting in energy-inefficient apps and shortened battery life. Hence, there is a need for a framework that can provide an end-to-end solution to address the challenges associated with detection, validation and repair of energy-inefficiencies related to energy bugs. In particular, such a framework should be able to address the following questions:

i. How to determine if an app contains an energy bug in a scalable fashion?
ii. How to generate test-cases that can demonstrate the presence of energy bugs in an automated fashion ?
ii. How to generate repair expressions that can fix the reported energy bugs?
iv. How to bring this (detection — test generation — repair) functionality to commonly used mobile-app development platforms such as Eclipse ADT?

Our framework, EnergyPatch, is the culmination of our effort to answer these questions. EnergyPatch extracts a model of the app (under test), using automated analysis. It then analyses this model using a light-weight, static analysis technique to detect program paths that may potentially lead to an energy bug. These potentially buggy program paths are then explored using a dynamic analysis technique to validate the presence of energy bugs. During exploration, if the presence of an energy bug is validated, EnergyPatch generates test-cases that bear witness to the presence of the reported energy bug. Finally, EnergyPatch, generates repair expressions for the reported energy bugs.

• *A. Banerjee, L. K. Chong and A. Roychoudhury are with the School of Computing, National University of Singapore, Singapore. E-mail: {abhijeet,cleekee,abhik}@comp.nus.edu.sg*

• *C. Ballabriga is with the University of Lille, France. E-mail: Clement.Ballabriga@lifl.fr*

1. One of the buggy applications we evaluated - Sensor Tester, drained a fully charged battery on a LG Optimus E400 smartphone in less than 8 hours whereas the standby time for the smartphone is approximately 600 hours [2]

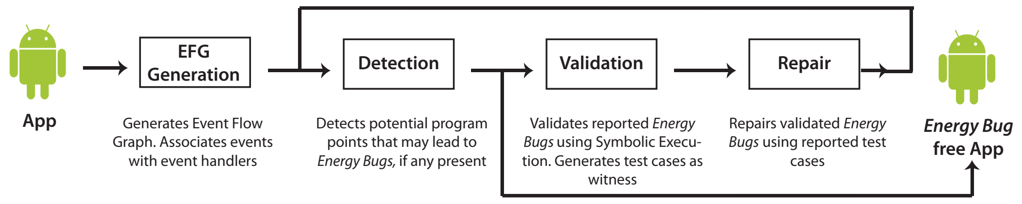

App
EFG Generation
Generates Event Flow Graph. Associates events with event handlers
Detection
Detects potential program points that may lead to Energy Bugs, if any present
Validation
Validates reported Energy Bugs using Symbolic Execution. Generates test cases as witness
Repair
Repairs validated Energy Bugs using reported test cases
Energy Bug free App

Fig. 1: System Overview

Figure 1 shows the three key phases in EnergyPatch: detection, validation and repair. More specifically, the detection phase is based on an abstract interpretation technique, the validation phase is based on a symbolic execution technique and the repair phase uses template based repair. It is worthwhile to know that using symbolic execution alone to explore non-trivial programs may lead to the problem of state-space-explosion. To ensure the scalability of our framework we use a multi-staged approach that is introduced in the following paragraphs.

To begin with, in the detection phase, our framework conservatively computes the set of program paths along which energy bugs may occur. If no such program paths can be found in the detection phase, we conclude that no energy bugs are indeed present in the app. It is worthwhile to know that results of the detection phase are always *sound* (*i.e.* if there is an energy bug in the tested app the detection phase will always report it). On the other hand, if the detection phase reports presence of program paths along which energy bugs may occur, we proceed to the validation phase. However, before executing the potentially buggy program paths using symbolic execution, we employ a couple of search-space reduction techniques to reduce the time required for exploration. These search-space reduction techniques are based on transitive closure computation and program slicing. Transitive closure computation (conservatively) determines which events (and corresponding event-handlers) in the app will never occur on potentially buggy program paths (as reported by the detection phase). Program slicing is used to (conservatively) estimate the subset of program inputs that do not influence the execution of the potentially buggy program paths. Finally, in the repair stage we use the information obtained from the previous phases to generate repair expressions for validated energy bugs.

We have implemented our framework as an Eclipse Plugin, named EnergyPatch. It is freely available from BitBucket [3]. Since a large number of Android app developers use Eclipse ADT Toolset for Android app development, we believe that our framework will be most useful in this form. Additionally, our tool provides an intuitive user interface that helps the developer in visualizing the potential energy bugs in the app. For the evaluation of our framework and tool we created a test-suite using thirty five real-life Android apps. Our framework was able to detect real energy bugs for twelve out of these thirty five tested apps. The test cases generated by our framework were manually executed on a mobile device and resultant power consumption measured by a power meter to confirm the presence of energy bugs. On measuring the energy consumption of the buggy apps post-repair we observed a reduction in energy consumption up to 60%. Finally, we conclude the evaluation of our framework by comparing it with existing research works on detection and/or test-generation for resource leaks in mobile-apps.

2 Background

Android is a widely-used operating system designed for mobile devices such as smartphones and tablets. Android is open-source, additionally there exist a number of freely available development tools (such as Eclipse ADT) and debugging utilities (such as Android Debug Bridge) for Android app development. All these factors motivated us to use Android apps as test subjects for our framework. In the following subsections we shall briefly describe some of the key aspects of execution model in Android and energy-inefficiencies observed in Android apps.

2.1 Execution Model in Android

Android apps, in general, are composed of four key components: *Activities*, *Services*, *Broadcast Receivers* and *Content Providers*. Communication between the various components happens by means of messaging objects that are commonly referred to as *Intents*. Figure 3 shows an oversimplified representation of the execution model for Android apps.

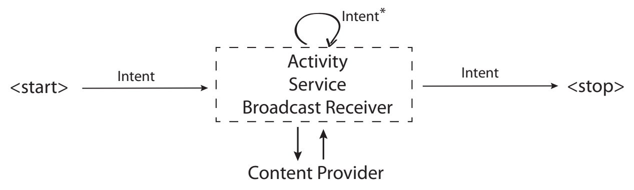
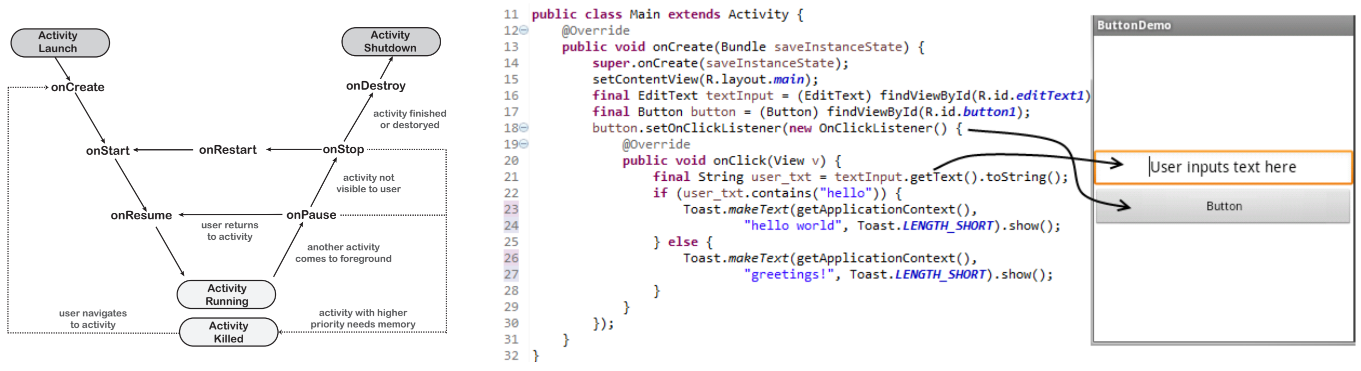


Fig. 3: Oversimplified representation of execution model in Android

An *activity* can be described as an entity which encapsulates the user interface (UI) through which the user interacts with the app. *Services*, unlike *activities*, do not have any UI elements associated with them. Services are run in the background and are often used for performing lengthy tasks. *Broadcast receivers*, as the name suggests, are used to receive broadcasted messages (such as arrival of call or SMS etc). *Content providers* provide access to various data sources. All components of an app have well-defined life-cycles. For instance, the life-cycles of an *activity* is shown in Figure 2(a).

An *activity* goes through seven distinct stages of life-cycle throughout its execution (*cf.* Figure 2(a)). The life-cycle stages shown in the left-arm of Figure 2(a) *i.e. onCreate*, *onStart* and *onResume* are invoked when an *activity* begins execution. Therefore, all tasks related to initialization and resource acquisition are usually performed in these stages. Likewise, the stages on the right-arm of the Figure 2)(a) *i.e. onDestroy*, *onStop* and *onPause* are invoked when an *activity* stops execution. In addition, the *onRestart* stage is invoked when a previously stopped activity is restarted. To implement a custom functionality to an *activity*, app-developers simply need to override the above mentioned methods (*i.e. onCreate*, *onStart*, *onResume*, *onDestroy*, *onStop*, *onPause* and *onRestart*).



```
11 public class Main extends Activity {
12     @Override
13     public void onCreate(Bundle saveInstanceState) {
14         super.onCreate(saveInstanceState);
15         setContentView(R.layout.main);
16         final EditText textInput = (EditText) findViewById(R.id.editText1)
17         final Button button = (Button) findViewById(R.id.button1);
18         button.setOnClickListener(new OnClickListener() {
19             @Override
20             public void onClick(View v) {
21                 final String user_txt = textInput.getText().toString();
22                 if (user_txt.contains("hello")) {
23                     Toast.makeText(getApplicationContext(),
24                             "hello world", Toast.LENGTH_SHORT).show();
25                 } else {
26                     Toast.makeText(getApplicationContext(),
27                             "greetings!", Toast.LENGTH_SHORT).show();
28                 }
29             }
30         });
31     }
32 }
```

(a) (b)

Fig. 2: (a) Lifecycle of an Android activity (b) a simple example showing how inputs are provided to Android apps

2.2 Inputs to an Android App

As shown in Figure 4, there are two possible ways of providing inputs to Android apps *(i)* through events and *(ii)* through return value from Android API calls. Events can be user-generated (such as by pressing buttons) or system-generated (such as broadcast of change in battery state). Figure 2(b) shows an example of input through an event as well as through return value. In the example of Figure 2(b), when the user clicks the button (event), the *onClickListener* (event handler) for the button is invoked(Line 18). Subsequently, the user inputs from the text field are read (input through return value), by means of another Android system call *getText* (Line 21). Observe that the else part of the example code is executed only if the return value from *getText* does not contain the string $hello$ (Line 22).

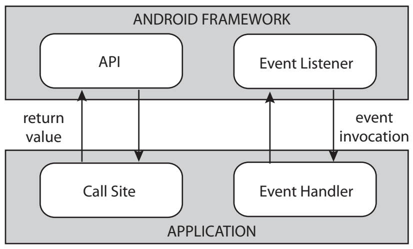


Fig. 4: Inputs to an Android app

This example goes on to show that *only exploring the event of the app is not sufficient to exercise all the functionalities in an app*. Based on this understanding we define an input to an Android app as follows:

Definition 1. Inputs to an Android app are a combination of events and return values from system call APIs.

2.3 Energy Consumption of Android System-call APIs

As is the case with any other non-functional property, energy consumption behaviour of a program is seldom explicitly encoded in its source-code. Therefore, in a generic scenario program analysis alone would be insufficient to determine the energy consumption behaviour of a program. However, in the specific case of Android apps we have identified a set of system call APIs that can substantially impact the energy consumption behaviour of the mobile device. In Android for instance, all power management utilities and I/O components must be accessed through predefined system call APIs. It is also worthwhile to know that these power management utilities and I/O components have significant impact on the power consumption of the device. Figure 5 shows the power profile graph of our test device $LG\ L3\ E400$. The data for Figure 5 is obtained from the `power_profile.xml` file which is created by the original equipment manufacturer (OEM) and shipped along with the mobile device. This file contains the average power consumption ratings for hardware components in the device, at different power states (such as for screen, power states are screen on and screen full). The data contained in this XML file is provided by the device manufacturer and therefore is reliable. Moreover, the Android framework uses this data to show battery related statistics. However, note that the data in this XML file is only an indicator of average power consumption of the hardware components of the device and does not correspond to any particular application being run on the device.

We found that in a typical Android distribution, the system call APIs that significantly impact the energy consumption behaviour are a very small subset of all available system call APIs. For instance, in Android Gingerbread, which was one of the most widely used distributions at the time of writing, less than hundred of the total nine thousand or so public system call APIs significantly affect the energy consumption behaviour. Some of these high-energy-consuming system call APIs are listed in Table 2.

3 Our Previous Work

In our previous work [1], we investigated various reasons for energy-inefficient behaviour in mobile apps. As a result of these investigations we were able to classify mobile-app related energy-inefficiencies into two categories: energy bugs and energy hotspots. The key difference between the two being that energy hotspots cause excessive power consumption while the (inefficient) app is under execution whereas energy bugs cause excessive power consumption even after the (inefficient) app has finished execution. As a result, in general, energy bugs cause more wastage of energy than energy hotspots. In the subsequent paragraphs, we recapitulate some of important aspects of our previous work. Finally, we end this section by highlighting the key differences between our current work and previous work [1].

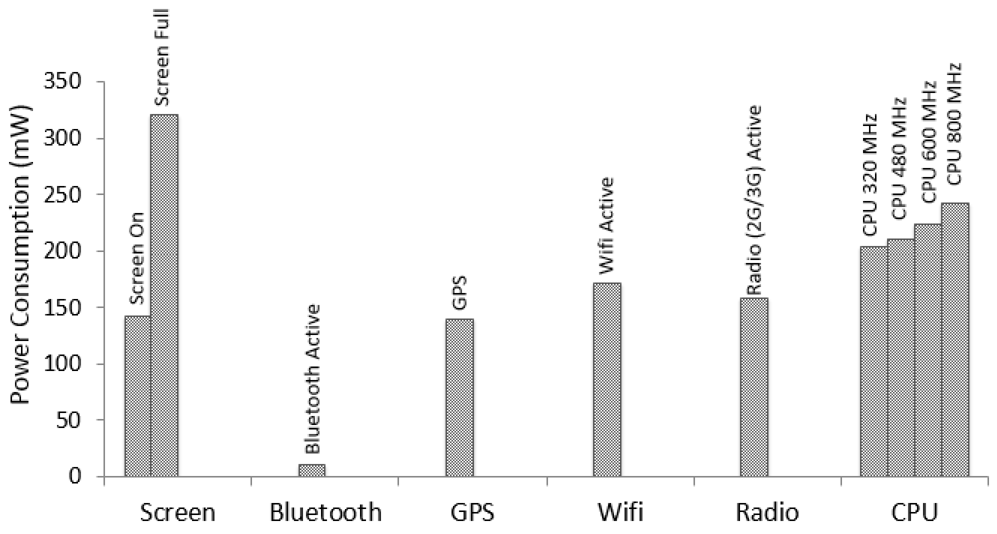


Fig. 5: Power profile for LG Optimus L3 E400 smartphone

TABLE 1: Various types of energy bugs that can occur in Android apps, as listed in our previous work [1]

#	Category	Type of Energy Bug
a	Hardware Resources	*Resource Leak*: Resources (such as the WiFi) that are acquired by an application during execution must be released before exiting or else they continue to be in a high-power state [4]
b	Sleep-state Transition Heuristics	*Wakelock Bug*: Wakelock is a power management mechanism in Android through which applications can indicate that the device needs to stay awake. However, improper usage of Wakelocks can cause the device to be stuck in a high-power state even after the application has finished execution. This situation is referred to as a Wakelock bug [5]
c	Background Services	*Vacuous Background Services*: In the scenario where an application initiates a service such as location updates or sensor updates but doesn't removes the service explicitly before exiting, the service will keep on reporting data even though no application needs it [6]
d	Defective Functionality	*Immortality Bug*: Buggy applications may re-spawn when they have been closed by the user, thereby continuing to consume energy [7]

TABLE 2: Some of the Android system call APIs that have major influence on energy consumption

Resources	**System Call APIs**	**Hardware Resource**
PowerManager (Wakelock)	acquire/release	CPU + Screen + Keypad
WifiManager	setWifiEnabled acquire/release	Wifi Hardware
Camera	open/close startPreview/stopPreview	Camera
SensorManager	registerListener/unregisterListener	Sensors
LocationManager	requestLocationUpdates/removeUpdates addProximityAlert/removeProximityAlert	GPS receiver
LocationClient	requestLocationUpdates/removeLocationUpdates addGeofences/removeGeofences	GPS receiver
MediaRecorder	start/stop	Video Hardware
AudioRecord	startRecording/stop	Audio Hardware
BluetoothAdapter	enable/disable	Bluetooth Hardware

3.1 Energy Bug, Cause and Effect

An energy bug is a type of non-functional defect (*i.e.* it does not affect the functionality of the app), however, it may cause the device to consume excessive power, even when no useful computation is being performed (*cf.* Definition 2). It is worthwhile to know that energy bugs are often a manifestation of improper usage of I/O components and power management utilities. There are two possible ways in which the effect of energy bugs can be handled.

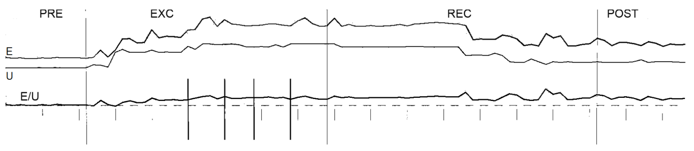
1) *Naïve Solution*: It is possible to construct a naïve solution (say using system-level mechanisms), that aggressively releases I/O components and power management utilities at all exit points of the app.

2) *Testing and patching*: It is possible to construct a systematic analysis framework that reveals scenarios (test-cases) where I/O components and power management utilities are not released on exit. The framework also suggests patches for each of this defect revealing scenarios.

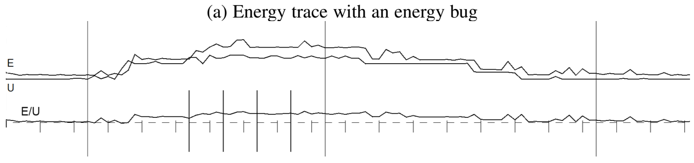
It is worthwhile to know that the naïve solution would be relatively simple to implement however such a solution might make the system (the platform on which the app executes), inflexible. This is because in certain cases it is possible that the app's functionality may necessitate that resources are not released as soon as the app exits. For instance, consider a location-tracking app that wants to log the user's whereabouts throughout the day. In such a scenario, the app may want to keep working in background using GPS while the user interacts with other apps in the foreground. For such apps, forceful system-level mechanisms that aggressively release resources whenever an app exits (*i.e.* the naïve solution), may make the system inflexible. In comparison, a case-by-case analysis using a testing and patching framework, such as the one presented in this work can help the developer in making judicious changes to the app source-code, wherever needed.

Definition 2. An energy bug can be defined as an energy-inefficiency that occurs when a mobile app does not release all energy-intensive components/resources acquired by it during execution, before it stops execution.

Figure 6(a) shows the energy-consumption data from a real-world Android app, Aripuca, while an energy bug is triggered. Figure 6(b) shows the energy-consumption behaviour of the same app, for the same input but when the energy-bug has been repaired. The portion of the energy-consumption trace marked with label PRE indicates the period of time when the device is idle (no apps running). Label EXC indicates the period of time where the app is executed on the device, whereas label REC indicates the period of time the device takes to return to its idle state. Finally, the label POST indicates the period of time when the device has returned to its idle state, post execution. In an ideal scenario, where there are no energy bugs involved, the energy-consumption behaviour in the PRE stage and the POST stage should be statistically similar. However, in the case of an app that has an energy bug (such as in Figure 6(a)), there will be significant dissimilarities in the energy-consumption data when comparing between the PRE and the POST stages.



(a) Energy trace with an energy bug



(b) Energy trace with the energy bug fixed

Fig. 6: Energy trace for *Aripuca GPS Tracker* (a) with an energy bug (b) repaired energy bug. The additional energy consumption can be observed in the recovery (REC) and the post (POST) stages

3.2 Classification of Energy-bugs

Table 1 lists the categorization of energy bugs that was described in [1]. The causes of energy bugs can vary from mismanaged resources, services, faulty sleep-state transition heuristics or defective functionalities. Such mismanagement can be attributed to developer carelessness (developer forgot to release resource after use) or unexpected execution flows (such as exceptions). Even if the app-developer were to somehow check all possible exit points of the app for the release of acquired resources, it would not be sufficient to ensure the absence of energy-bugs in the app. Consider the code snippet shown in Figure 7. This simple code is supposed to start a location-update background service (Line 10) in the $onCreate$ method. Subsequently, it executes some unrelated functionality in (Line 14). When the user navigates away from the application the event handler $onPause$ is executed. In the method $onPause$ location-update service is removed (Line 22). However, if there is an exception before Line 22 (for instance, in this example, due to Line 21), the location update service is never stopped and execution can result in an energy bug scenario. This example demonstrates that even if developer encodes the code to release resource at the exit point energy bugs may occur.

```
1 LocationManager locationManager;
2 long Min_Update_Time = 10, Min_Distance = 1000 * 60 * 1;
3
4 @Override
5 public void onCreate(Bundle savedInstanceState){
6      super.onCreate(savedInstanceState);
7      setContentView(R.layout.main);
8      locationManager = (LocationManager)getSystemService
9                                           (LOCATION_SERVICE);
10     locationManager.requestLocationUpdates
12     (LocationManager.GPS_PROVIDER,Min_Update_Time,
13                              Min_Distance, this);
14     someOtherFunctionality();
15 }
16
17 @Override
18 public void onPause(){
19     super.onPause();
20     try{
21            functionMayThrowsException();    <---------
22            locationManager.removeUpdates(this);
23     }catch(Exception ex){
24            Log.v(”test”,”exception occured”);
25     }
26 }
```

Fig. 7: Code fragment with a potential energy bug

3.3 Differences Between Present and Previous Work

The key contributions of our previous work [1] were the introduction of a fault-model for energy-inefficient behaviour in mobile apps and a guided, search-based test-generation framework. The energy-inefficiency detection in [1] was done using a hardware-software, hybrid approach. Like our current work, it used an automated technique [8] to generate the event-flow graph of the app (under test). However, unlike our present work, it uses a guided, search-based algorithm to select the test-inputs to execute the app on the target mobile device. While the app is being executed on the test device (with the selected test-input) the framework simultaneously measured the power consumption using a power meter. The acquired power trace is then analyzed using statistical and anomaly detection techniques to uncover energy-inefficient behaviour. In contrast, in our current work, we use a combination of a static and dynamic analysis techniques to ascertain the presence of energy bugs and to generate test-cases. The use of measurement setup (*cf.* Figure 22), in our present framework, is simply to measure the resultant energy savings. Another key difference is in the way of exploration itself. In the previous work, the exploration algorithm generates events traces, by walking through the event flow graph. However, if an UI screen needed inputs through an input-container (such as text-fields), random data was used. As a result, the exploration algorithm may not have been able to explore all feasible paths inside an event-handler. In contrast, in our present work, due to the use of symbolic execution our framework can explore all feasible paths inside the event handler. To summarize, the key differences between the two work arise due to (i) the way energy bugs are detected (power measurement vs static analysis) (ii) the way test-cases are generated (search heuristics vs guided symbolic execution) and (iii.) automatic repair expressions generation (only in the current framework).

4 Overview by Example

In this section, we shall describe the workings of our framework by means of a simple example. In particular, we shall focus on (i) use of abstract interpretation to detect energy bugs and (ii) use of symbolic execution to generate test cases that leads to the reported energy bugs. In our framework, we also use an event-flow graph (EFG) generation phase as a pre-processing phase. However, for the purposes of simplicity, we shall omit the EFG generation phase in this example. We shall base our discussion on the simple code fragment shown in Figure 8(a). It has a simple input-dependent, do-while-loop, in which, a source line acquires a reference-counted resource R at Line 4 and another source line releases the acquired resource at Line 6. The only input to the program is N, which determines the number of loop iterations. Figure 8(b) shows the control flow graph (CFG) of the code shown in Figure 8(a). It can be observed that the resource R is never released for all inputs satisfying the formula $N > 3$. Using this example, we shall first describe how static analysis is used in our framework to detect that the resource R may not be released at the end of the program (*i.e.* exit node $E4$). Subsequently, we will describe the use of dynamic analysis to generate test cases that witness the scenarios where the R hasn't been released.

4.1 Detection Using Abstract Interpretation

The detection phase tracks an (over)estimate of the state of resource R, at each program point. Assume that the state of resource R is denoted by a tuple $<R, k>$ at each node of the graph (cf. Figure 8(c)). The input (in) and output (out) state of resource R are shown using a tuple $<R, k>$ at each node of the graph in Figure 8(c), where $k \leq 0$ implies R is *not acquired* and $k \geq 1$ implies R is *acquired*. Every time the system call Acquire(R) is encountered the resource state is $updated$ as shown in Equation 1, Acquire(R). Similarly, whenever the system call Release(R) is encountered the resource state is $updated$ as shown in Equation 1, Release(R). It is worthwhile to know that resource state update operation is *object-insensitive*.

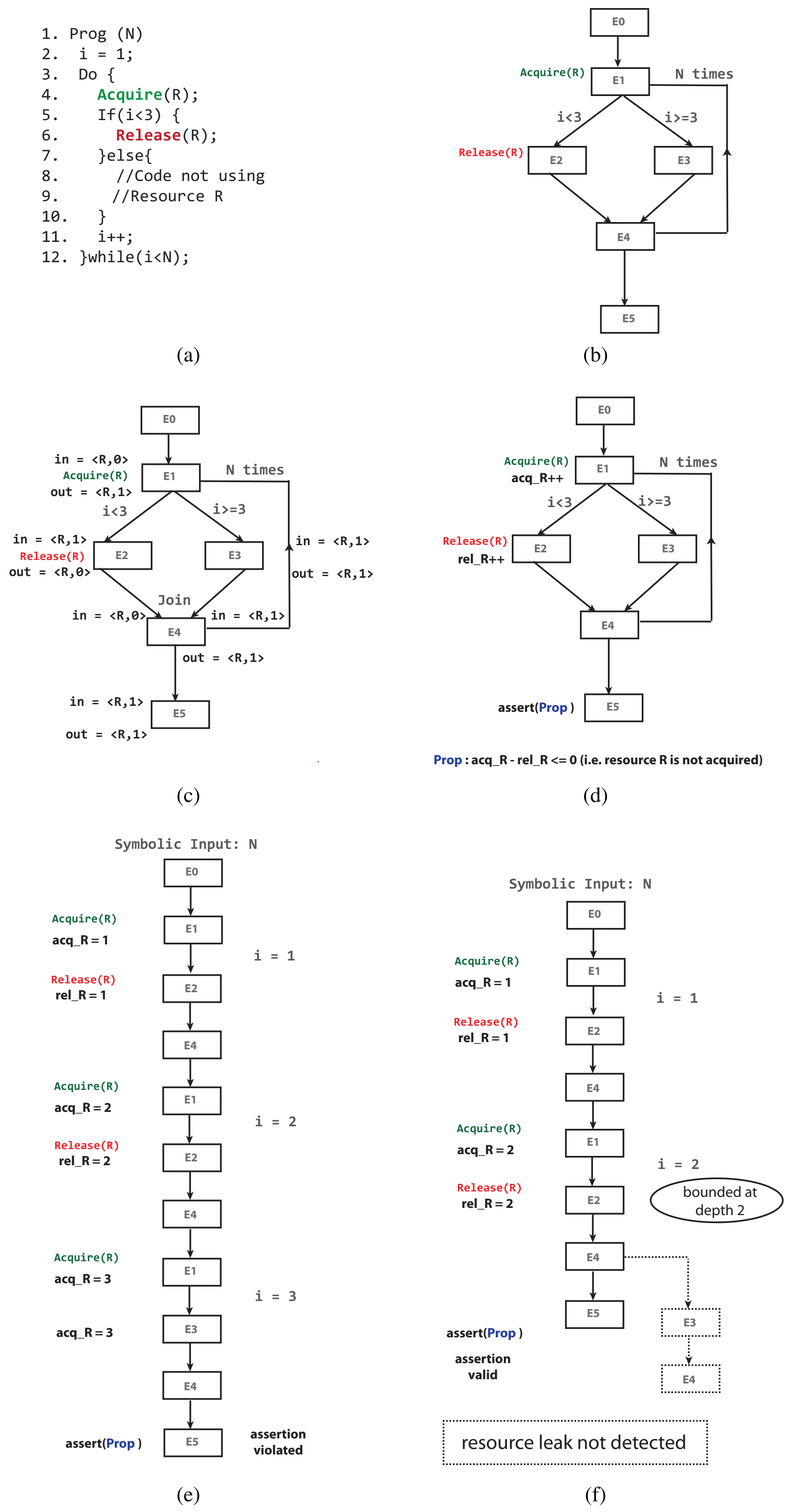
$$Update(<R,k>) = \begin{cases} <R,k+1>, & \text{Acquire(R)} \\ <R,k-1>, & \text{Release(R)} \end{cases} \tag{1}$$

In the scenario where there are multiple incoming resource states (from different branches) we perform a $Join$ operation to merge multiple resource states into one as shown in Equation 2. For instance, in the example of Figure 8(c), at the node marked $Join$ the incoming resource states from both the branches ($<R, 0>$ from $i < 3$ branch and $<R, 1>$ from $i \geq 3$ branch) are joined to create an over-approximated state ($<R, 1>$). By performing such path-insensitive joining of states, we can avoid the problems associated with state-space explosion.

$$Join(<R,k_1>,<R,k_2>) = \begin{cases} <R,k_1>, & \text{If } k_1 > k_2 \\ <R,k_2>, & \text{Otherwise} \end{cases} \tag{2}$$

The $Update$ and $Join$ operations that are described here, are applied until the in and the out states of each node in the graph do not change over an entire iteration (*i.e.* fixpoint is achieved). At the end of detection phase (once the fixpoint is reached), we check the out state of exit nodes to see if any resource is still in the $acquired$ (*i.e.* $k \geq 1$) state. If so, presence of potential resource leak is reported by the detection phase. In our example (*cf.* Figure 8(c)), observe that at the exit node, the resource state is indeed $<R, 1>$, indicating the presence of a resource leak. This leak will manifest for all inputs satisfying the condition $N \geq 3$.

It is worthwhile to know that the detection phase has the property of scalability due to its path-insensitive nature. More specifically, the resource states ($<R, k>$) themselves do not carry any path specific information. Such as in Figure 8(c), no path-specific information (such as $i < 3$ or $i \geq 3$) is carried within the resource states along the if/else branch. Also the $Update$ and $Join$ operations (as shown in Equations 1 and 2) do not take any path-specific inputs. Such path-insensitivity in the detection phase does provide scalability, but may lead to overestimation. In particular, the detection phase is more likely to produce overestimated results in scenarios where the control flow of the program contains infeasible paths. For instance, *if* the path satisfying the condition $i \geq 3$ in Figure 8 had been infeasible, the results of the detection phase would have had a false positive. Therefore, to rule out any possibility of false positives, we subsequently use a path-sensitive dynamic exploration technique to validate the results generated by the detection phase. Essentially, we wish to validate that the property (*Prop: resource R is not acquired*) at all exit nodes. To facilitate this, our framework automatically instruments two new variables acq_r and rel_r in the program. Specifically, wherever Acquire(R) is called variable acq_r is incremented and wherever Release(R) is called variable rel_r is incremented. Additionally, at the exit node the assertion $acq\_r - rel\_r \leq 0$ is instrumented that represents the property resource R is not in the acquired state. The resultant CFG is shown in Figure 8(d).



```
Prog (N)
 i = 1;
 Do {
   Acquire(R);
   If(i<3) {
     Release(R);
   }else{
     //Code not using
     //Resource R
   }
   i++;
 }while(i<N);
```

Fig. 8: Overview by example (a) example code with a potential resource leak (b) CFG othe the example code (c) static analysis of the code. Input and output abstract states are shown for each node in the graph (d) assertion added to the exit node of the graph (e) symbolic exploration and test case generation (f) limitation while using bounded symbolic execution

4.2 Test Generation Using Symbolic Execution

The instrumented program is then explored symbolically. That is to say that the program is executed with symbolic inputs. In our example symbolic input being N. In our framework, we also perform a couple of search-space-reduction techniques to make symbolic execution phase faster. However, for the sake of simplicity we shall not describe them in this example.

The objective of symbolic execution of the instrumented program is to see if any of the instrumented assertions ($acq\_r - rel\_r \leq 0$, in this example), are violated for any feasible execution. For instance, in Figure 8(d) if $N = 1$, a feasible execution would involve going through basic blocks E0, E1, E2, E4 and E5 (in that particular order). Another example of feasible execution for $N = 3$ is shown in Figure 8(e). It is interesting to note that in this particular example, Figure 8(e) shows a possible execution path along which the instrumented assertion is violated. It is also worthwhile to know that if we had used bounded symbolic execution instead of unbounded symbolic execution for exploration, we may not have been able to find such an assertion violation (as shown in Figure 8(f)). Assertion violation demonstrates a feasible scenario where a resource leak happens in this example. The symbolic execution also provides us with the test-cases that witness the failure of assertion. These test-cases can be used by the developer to re-create the reported bugs manually.

5 EFG GENERATION

Mobile applications, such as Android apps, are event-driven applications. Even though many Android apps are developed using JAVA they do not contain a *main* method like conventional JAVA programs. Instead, such apps usually consist of several disconnected pieces of code called event handlers. Each event handler is responsible for processing of a specific event. Ordering between two instructions of an event handler is represented by means of a control flow graph (CFG). The CFG however does not provide any information regarding the ordering between any two event handlers. To represent the ordering between any two event handlers in the application we use an event flow graph (EFG). The EFG we use in our work is an extension of the EFG proposed in [9]. The key difference being that in our EFG the nodes also contain the CFG of the event handler. Figure 9 shows a pictorial depiction of an EFG. *The key purpose of the EFG is to provide a model for the inter-component communication in the app.*

Definition 3. An Event Flow Graph (EFG) is a directed graph, capturing all possible event sequences that might be executed for an event-driven application. Nodes of an EFG represent events and contain the control flow graph (CFG) of the respective event handler. A directed edge between two EFG nodes X and Y represents that event Y can follow event X.

Given the class files for an app, our framework can generate the CFGs for all event handlers (and other methods) within the app. To generate the EFG of an app we use a modified version of Dynodroid [8] to automatically explore the GUI of the app and generate its EFG. Dynodroid itself relies on the Android tool Hierarchy viewer to obtain information about the user interface layout of the app. It uses this information to crawl through the graphical user interface of the app. We further modified Dynodroid to progressively generate the EFG of an app. The EFG generation process is illustrated in Figure 10. Additionally, we need to associate the event-handlers (CFGs) to the events (EFG nodes). To do so, we instrument all event-handlers in the app before exploration such that all invocation of event-handlers are recorded to a log file. Additionally, all invocation of events (by Dynodroid) are also recorded to the same log file. Since all entries in the log file are accompanied by their timestamps, we can easily associate each event to its event-handler.

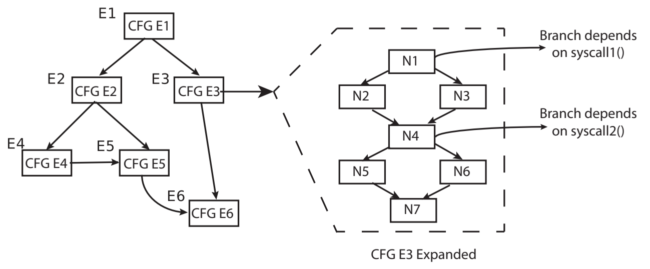


Fig. 9: An example event flow graph (EFG) with events E1 through E6. Each event has an attached control flow graph (CFG). Here the CFG for event node E3 has been expanded. Nodes N1 through N7 represent the control flow nodes for the event handler that is represented by E3

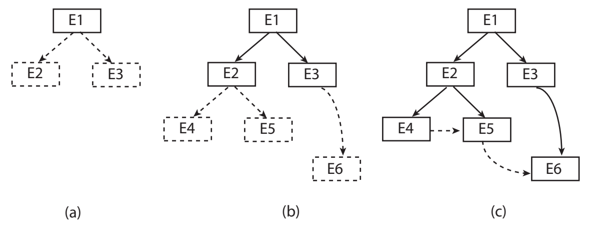


Fig. 10: An illustrative example for EFG generation. The dashed lines/boxes represent the EFG edges/nodes that have been discovered but not yet explored, while the solid lines/boxes represent the EFG edges/nodes that have been discovered. The EFG edge/node information is obtained through Android Hierarchy Viewer whereas the exploration is done using Dynodroid

It is worthwhile to know that the EFG generated by Dynodroid may be incomplete. However, our tool allows the developer to manually augment the EFG. Since our tool is specifically targeted at the developer, who has the best knowledge of the GUI model of the application, it can be assumed that the GUI model of app obtained after the EFG Generation phase would be adequate.

6 DETECTION

In this section, we shall provide a detailed description of the detection phase. As mentioned in Section 4, the static analysis technique used in the detection phase is an instantiation of the abstract interpretation based approach proposed in [10]. Therefore, before describing the details of our approach we shall provide a very brief introduction to abstract interpretation itself.

6.1 Abstract Interpretation

To implement an abstract interpretation based program analysis framework, one needs to define the abstract semantics, in particular, (a) an *abstract domain* and (b) a set of *abstract operations*. At each program point (P) an abstract state captures the state of the program. The set of all abstract states is referred to as the abstract domain (D). The *abstract operations*, namely *update* and *Join*, can be used to manipulate the abstract state to reflect the effect of execution (on the property of interest) along a program path. More specifically, the *update* operation reflects the effect of executing an instruction over the abstract state and has the type of Equation 3.

$$U : D \times P \rightarrow D \tag{3}$$

The abstract join operation (J), on the other hand, is used to combine two abstract states into one. In our implementation, the join operation computes the least upper bound on the abstract domain (D). Whenever there are multiple abstract states coming from different control flows into a program point, we use the abstract join operation to combine them. The abstract join operation has the type of Equation 4.

$$J : D \times D \rightarrow D \tag{4}$$

6.2 Instantiation of Abstract Interpretation for our Framework

The property of interest in our framework is shown in Property 1. A resource can be present in either of these two states : *acquired* or *not acquired*. Furthermore, if a resource is reference-counted, it is associated with an integer that is incremented whenever the resource is acquired, and decremented whenever the resource is released. If there are one or more resources that are in the *acquired* state at the end of the detection phase we report the potential for an energy bug.

Property 1. *All energy-intensive, hardware resources and power-management utilities should be in the* released *state at all exit nodes of the (analyzed) app.*

Resources in the Android applications are represented by Java objects. Therefore, we shall first define the semantics for Java object tracking in subsection 6.3. Subsequently, we shall extend this representation for resource tracking in subsection 6.4.

6.3 Java Object Tracking

To reliably track Java objects, we need to have a domain D, that abstracts the various memory structures containing objects. We will need to represent (1) the set of object references (O), (2) the Java stack (s) and (3) the variables (v).

1. Let $O = J \cup \{\top\}$ be the set of abstract object references. The set J represents the set of concrete Java object references. Element $o \in O$ either represents a concrete Java object, or is equal to $\{\top\}$ (any Java object).
2. Let $s : \mathbb{N} \rightarrow O$ be a function representing an abstract stack state, and let S be the set of abstract stack states. For any stack state s, the value $s(0)$ represents the top-most (most recently pushed) element, the value $s(1)$ represents the element pushed before $s(0)$ and so on.
3. Let $v : E \rightarrow O$ be a function representing the abstract variable states, and let V be the set of such states. The set E represents the set of possible Java variable expressions. An element in E can refer to a local variable, a member field, or a static field.

The abstract domain (D) for tracking Java objects is of type as shown in Equation 5, where $\mathcal{P}(O)$ represents the power set of all abstract object references, S represents the abstract stack state and V represents the abstract variable states.

$$D = \mathcal{P}(O) \times S \times V \tag{5}$$

6.4 Resource Tracking

In order to track a resource object we need to extend the domain of D with the state of the resource and the set of possible acquire locations. The resultant abstract domain D' is shown in Equation 6, where K represents all possible states for a resource and P represents the set of all program points. A resource can either be in *acquired* or *not acquired* state. If the resource is reference counted, its state is equal to the upper bound on the acquire count, or to the value $+\infty$, if it cannot be statically bounded. Therefore, the resource state K equals the set $\mathbb{N} \cup +\infty$.

$$D' = D \times K \times P \tag{6}$$

Now since our abstract domain has been defined we can further elaborate on the nature of the abstract operations for resource tracking. The update operation (U') at a program point P, can be represented by Equations 7 and 8. Here $d' \in D'$ and $d' = \langle d, k, p \rangle$, where $d \in D$, $k \in K$ represents the state of the resource and p is the set of acquire locations.

$$U' : D' \times P \rightarrow D' \tag{7}$$

$$U'(d', P) = \begin{cases} U \bullet U_{res}(d', P) & \text{, if instruction at } P \text{ is an acquire} \\ & \text{or release instruction} \\ (U(d, P), k, P) & \text{, otherwise} \end{cases} \tag{8}$$

U denotes the abstract update operation for an instruction that is not related to resource acquire or release. The symbol $\bullet$ denotes function composition. The operation U_{res} in Equation 8 is invoked whenever a resource acquire or release instruction is encountered. The function of U_{res} is as follows. Whenever we encounter an instruction for acquiring a resource r, we add P to the set of acquire locations for r. Additionally, if the resource r is reference counted we increase $k_r (\in k)$ by one, otherwise we set k_r to 1. On encountering a release instruction, we reduce $k_r (\in k)$ by one, if $k_r > 0$.

The Join operation (J) can be represented by Equations 9 and 10, where we join resources from two sets ($D_1, D_2 \in D'$). If both sets contain the same resource (*i.e.* associated with the same Java object), we take the maximum of the reference counts (Max operation in Equation 10), and we merge the acquire-location sets. For all other cases (*i.e.* when $d_1 \neq d_2$) we abstract the Java object to *top* ($\top$ represents the largest element) and add to the resultant set.

$$J : D' \times D' \rightarrow D' \tag{9}$$

Let $D_1 = \langle d_1, k_1, p_1 \rangle$ and $D_2 = \langle d_2, k_2, p_2 \rangle$.

$$J(D_1, D_2) = \begin{cases} \langle d, Max(k_1, k_2), p_1 \cup p_2 \rangle & \text{if } d1 = d2; \\ \langle \top, k_1, p_1 \rangle \cup \langle \top, k_2, p_2 \rangle & \text{otherwise.} \end{cases} \tag{10}$$

6.5 Detecting Potential Energy Bugs, Instrumenting Assertions

As a result of the abstract interpretation based analysis we can get the abstract state at each program point. To detect the presence of energy bugs we are particularly interested in the abstract state at the end of the program. An Android app can have many activities; the user can exit the app at any activity. However, whenever the user exits the app the activity lifecycle stage (and event handler) *onPause* is invoked. Therefore, all *onPause* event handlers within the activities contained in an app can potentially be the end of the program. To simplify our analysis, we construct an hypothetical EFG node END (denoting the end of the program) and connect it to all those EFG nodes that are associated with *onPause* event handler with a uni-directional edge (the direction of the edge being *onPause* EFG nodes *to* END node).

Let $\langle d, k, p\rangle \in D'$ be the abstract state at the end of the program. Then k represents the state of resources at the end of the program. An abstract state at the end of the program with $\exists(k_r \in k), k_r > 0$ denotes that resource r that may have been acquired but not released on some path in the program. In other words Property 1. is violated. Such a scenario implies the presence of a potential energy bug.

To detect resource leaks, our framework automatically instruments Property 1. as assertions at the exit node of the (analyzed) app. The exact instrumentation slightly differs for reference counted and non-reference counted resources. However, in the both cases we first instrument two new counter variables acq_r and rel_r for each (potentially) unreleased resource r. The instrumentation is such that the variable acq_r is increased every time resource r is acquired and variable rel_r is increased every time resource r is released. For a reference counted resource, the assertions is such that it checks the value for formula $acq\_r - rel\_r \leq 0$ *i.e.* there are at least as many releases as acquires for the resource r. Whereas for a non-reference counted resource the assertions checks the value for formula ($acq\_r \neq 0$) $\Rightarrow rel\_r > 0$ *i.e.* if a resource is acquired, there are at least one or more releases for the resource. Once instrumented these assertion are tested for violations in the validation phase.

A note on implementation of instrumentation in the framework: It is worthwhile to know that in the framework implementation, the instrumentation of counter variables acq_r and rel_r is done in the library (instead of the app code) where the Acquire(R) and Release(R) are defined respectively.

A note on detecting resources leaks in components of an app: As shown in Figure 3, an Android app can be composed of components such as activity, services, receivers and content providers. In Section 6.5 we have discussed resource leak detection only in context of an activity. However, the technique described in Section 6.5 can be extended to other (non-activity) components of an app, provided the end stage event-handler for these components are known. This information is readily available from the Android Developers website. For instance, the end stage event-handler for a service component would be *onDestroy* [11].

7 Validation

The potential energy bugs detected in the previous phase are validated in this phase. In this phase, we use a symbolic execution based technique to test the assertions instrumented in the previous phase. It is worthwhile to know that symbolically executing the entire application may often be impractical due to the issue of *state-space explosion*. Therefore, before symbolically exploring a potentially buggy application we apply a couple of search space reduction techniques to reduce the number of program states that need to be explored (in ordered to validate or invalidate the instrumented assertions). The search-space reduction techniques, namely (a) transitive closure computation of EFG and (b) symbolic input reduction, are discussed in Section 7.1. Subsequently, test input generation process is detailed in Section 7.2. The complete flow of the validation phase is shown in Figure 11.

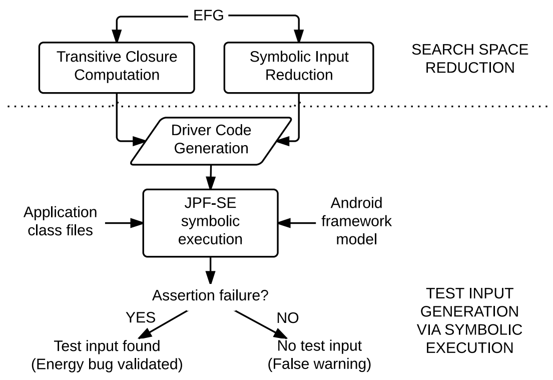

EFG
Transitive Closure Computation
Symbolic Input Reduction
SEARCH SPACE REDUCTION
Driver Code Generation
Application class files
JPF-SE symbolic execution
Android framework model
Assertion failure?
YES
NO
Test input found (Energy bug validated)
No test input (False warning)
TEST INPUT GENERATION VIA SYMBOLIC EXECUTION

Fig. 11: Overview of the validation process

7.1 Search Space Reduction

Transitive closure computation of EFG: The event flow graph (EFG) of an application captures all events in an application. However, some of the events (represented by nodes in the EFG) may not influence the acquiring of a resource in any feasible execution. Therefore, such events (EFG nodes) can be excluded during exploration. The nodes that need to be explored are grouped into two sets. The first set (S_1) consists of all nodes that fall on a path from an entry node to the resource acquiring node. The second set (S_2) consists of all nodes that fall on a path from the resource acquiring node to an exit node. All nodes that are not contained in these two sets (*i.e.* $S_1 \bigcup S_2$) need not be explored symbolically. This computation is repeated for all resources that may lead to potential energy bugs (as reported in detection phase). Figure 12 shows an example of transitive closure computation. In Figure 12(a) there are three paths from entry node ($E1$) to exit node ($E6$), however only the path $E1 - E3 - E6$ is of interest for checking the validity of Property 1.

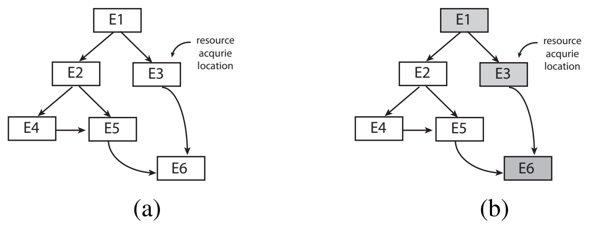


Fig. 12: Example of transitive closure computation. EFG node $E3$ is resource acquire location. Transitive closure computation gives the list of nodes shown in shaded in (b)

Symbolic input reduction: Techniques based on symbolic exploration often face the issue of scalability whenever there are a large number of symbolic program states to be explored. The number of symbolic states to be explored is directly influenced by the number of symbolic inputs to the program. Since Android applications are *event-driven*, inputs to an application may arrive during execution. As shown in Figure 4, the two potential inputs for an Android application are *(i)* the return value of an Android API call, and *(ii)* the arguments supplied to an application-level event handler when the Android system invokes the callback routine (due to an event trigger). A typical Android application may receive many such inputs (*e.g.* the application may frequently invoke Android

Algorithm 1 Slicing for relevant inputs in a program

```
1: Input:
2: $R$: resource that may be involved in an energy bug
3: Output:
4: $V$: the set of input variables to be made symbolic
5:
6: $T \leftarrow \{\}$
7: for all acquire sites of resource $R$, $I_{acq}$ do
8:     $S \leftarrow$ compute backward slice w.r.t. instruction $I_{acq}$
9:     $T \leftarrow T \cup S$
10: end for
11: for all release sites of resource $R$, $I_{rel}$ do
12:     $S \leftarrow$ compute backward slice w.r.t. instruction $I_{rel}$
13:     $T \leftarrow T \cup S$
14: end for
15: $V \leftarrow$ ParseSlice($T$)
16:
17: function ParseSlice($slice$)
18:     return set of all input variables that are used in any instruction contained in $slice$
19: end function
```

```
1 if (i1 == 0) {
2   if (i2 == 0) {
3       v1 = 0;
4   }
5   if (i3 == 0) {
6       v2 = 0 ;
7   }
8   acquire(R);
9   if (v1 == 0) {
10      release(R);
11  }
12 }
13 else {
14  if(i4 == 0) {
15      release(R) ;
16  } else {
17      v1 = 0 ;
18  }
19 }
```

Set of relevant inputs: i1, i2

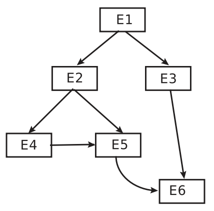
Fig. 13: An example showing how our slicing algorithm (Algorithm 1) works. The bold lines (figure on the right) shows the slice after application of Algorithm 1. The inputs to this example program are `i1`, `i2`, `i3` and `i4` while `v1` and `v2` are local variables. For the resource to be acquired and not released, only input variables `i1` and `i2` (source code lines numbers $1, 2, 3, 8, 9$ and 10) are relevant. It is worthwhile to note that the entire else branch (lines 13 - 19) is irrelevant as no resource is ever acquired if the execution comes to the else branch.

APIs and read the return values). As the values of the inputs are not known statically, we have to treat the input values as symbolic during exploration. However, exploring the program with all input variables made symbolic may be very expensive (or even impractical). To alleviate this issue we selectively make an input variable symbolic only if that input-variable may affect the execution of program paths where the potential resource leaks are reported.

To implement this, we use an existing *static program slicing* technique to capture the set of instructions in a program that may influence the execution of the program towards an energy bug, due to a resource R. Specifically, for each acquire(release) site of resource R, we statically compute a *backward slice* with respect to the instruction that acquires(releases) R. Subsequently, only the input variables that are used by any instruction captured in any of the computed slice have to be made symbolic. Figure 13 shows a simple example of slicing algorithm shown in Algorithm 1.

7.2 Test Input Generation

The final step in the validation phase is the generation of test inputs that expose assertion violation (and hence energy bugs) in the analyzed app. For this purpose we use the tool JPF-SE [12], which is a symbolic execution extension for the tool Java PathFinder (JPF). It is worthwhile to know that Android apps, unlike conventional Java programs do not have a *main* method as the starting point of execution. Instead the execution of an Android application starts from a *root* UI screen. JPF-SE however works for conventional JAVA programs only, therefore our framework automatically generates a driver file that represents the structure of the analyzed app's EFG. The generation of the driver code is a straightforward process. The first event handler to be called in the driver is that of the *root* UI screen, followed by its child nodes (event handlers). In the scenario where an EFG node E contains multiple child nodes $c_1, c_2, \ldots, c_i$, we create conditional branch statements for each child node c_i. The execution of a conditional branch statement is decided based on a newly added variable $ctrl\_E$. Essentially, the variable $ctrl\_E$ represents the event (or user input) that decides the execution of a child node at E. While executing the application symbolically, we make the variable $ctrl\_E$ symbolic. This allows us to explore all possible event sequences at a given EFG node in the application. Figure 14 shows an example for driver code generation.



```
void E1() {
     /* Invoke E1's event handler here */
    if (ctrl_E1 == 0) {
          E2();
    } else {
          E3();
    }
}
void E2() {
    /* Invoke E2's event handler here */
    if (ctrl_E2 == 0) {
          E4();
    } else {
          E5();
    }
}
. . .
```

(a) (b)

Fig. 14: An example for driver code generation

As described in Section 6.5, for all potential energy bugs reported by the detection phase, we instrument assertions at the exit points in the EFG. Symbolically executing through the application via the driver code allows us to check the validity of the instrumented assertions. Each assertion violation is recorded and the corresponding failure revealing test-cases is presented to the developer as a witness for the reported bug.

Following sequence provides an example of a bug-revealing test sequence generated by our framework for the app Tachometer. The bug-revealing test sequence contains all the information that the developer needs to replicate the reported bug. For instance, in the following example the bug-revealing scenarios tells about the UI events and their relative ordering that needs to be triggered to observe the reported bug. In addition, the framework also reports the event-handler signatures (shows in square brackets in the following example), to further assist the developer. It is worthwhile to know that a single user event can trigger multiple event-handlers, such as in the following example, event `id/button1/TAPSCREEN_120_93` triggers event-handler `WahlActivity$1_onClick` followed by `PositionActivity_onCreate`.

```
Buggy Sequence:
  entryNode/KEYPRESS_82 [WahlActivity_onCreate]
  -> id/button1/TAPSCREEN_120_93 [WahlActivity$1_onClick]
  -> [PositionActivity_onCreate]
  -> MenuButton/KEYPRESS_82/
  -> BackButton/KEYPRESS_4/
  -> [PositionActivity_onPause]
```

Fig. 15: Test sequence generated for app *Tachometer*

It is worthwhile to know that *all the bug revealing test cases generated by our framework are valid*. This is due to use of a dynamic exploration technique for test-generation.

8 Automated Repair

In the final phase, our framework automatically generates repair expressions for the validated energy bugs. Figure 17 shows the work-flow of this final phase. To generate a repair expression, we need to determine (i) the repair expression and (ii) the repair location. The repair expression is affected by the choice of repair location and hence we first discuss how the repair location is obtained.

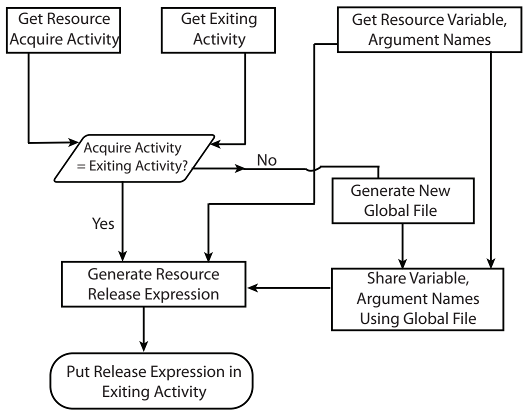

Get Resource Acquire Activity
Get Exiting Activity
Get Resource Variable, Argument Names
Acquire Activity = Exiting Activity?
No
Yes
Generate New Global File
Generate Resource Release Expression
Share Variable, Argument Names Using Global File
Put Release Expression in Exiting Activity

Fig. 17: Work flow for automated repair in our framework

It is worthwhile to mention that the objective of the repair phase is primarily *to fix the reported energy bug* (*i.e.* to modify the source code such that the bug-revealing test-case no longer triggers a bug). However, in some scenarios, there might be multiple locations at which the repair expression can be added. For example, in the simplistic scenario shown in Figure 18, the repair code can be either put at $RepairLocation1$ or at $RepairLocation2$, depending on whether or not $Activity$ 2 needs the resource acquired in $Activity$ 1.

To be energy-efficient an acquired resource should be released as soon as it is not required any more. However, determining whether an acquired resource is still needed may not be feasible just by analysing the instructions in the application bytecode. This is because for certain resources, such as Wakelocks, *last-use* information is not explicitly found in the application code (a Wakelock prevents the CPU from going into sleep-state as long as it is acquired). Therefore in our framework, the repair expression is always put in the last method ($onPause$) of the exiting activity in the bug-revealing test case (generated during the validation phase). Here exiting activity refers to the (sequentially) last activity in the bug-revealing test-case. Our strategy is always guaranteed to fix the energy bug as witnessed by the bug-revealing test case, however the automatically generated repair may not be optimal under all circumstances.

$$<\text{resource\_expr}> . <\text{release\_system\_call}> \qquad (11)$$

The repair expression automatically generated by our framework has the format shown in Equation 11. The resource_expr part in Equation 11 represents the expression to access the resource object at the repair location and release_system_call represents the system call API to release the resource object. To form a syntactically correct repair expression we need to obtain the variable name for the resource object and the arguments to the release system call. These information are obtained from the result of the detection phase (described in Section 6).

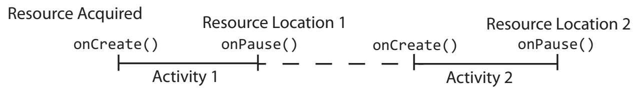


Fig. 18: An example scenario

In the scenario where resource acquiring activity and exiting activity are different (such as in the example of Figure 18), our framework adds additional pieces of code to ensure syntactic correctness. In particular, a new global file is automatically added

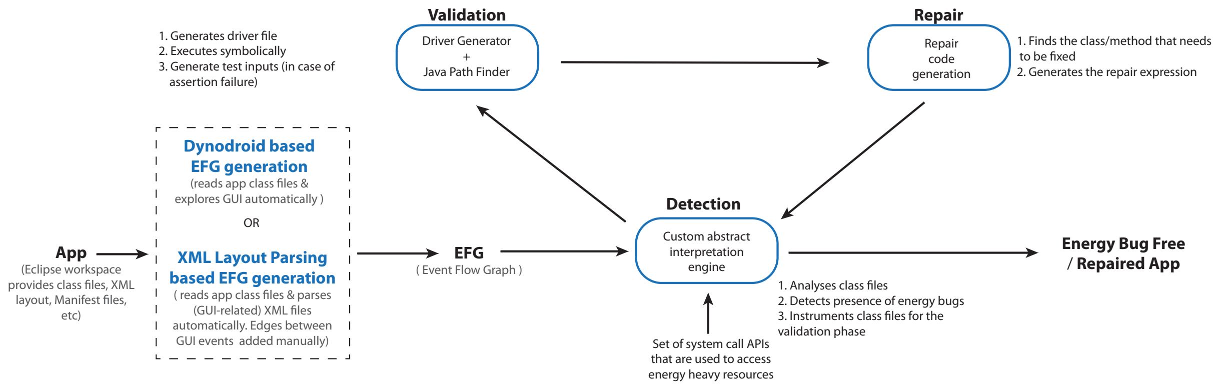
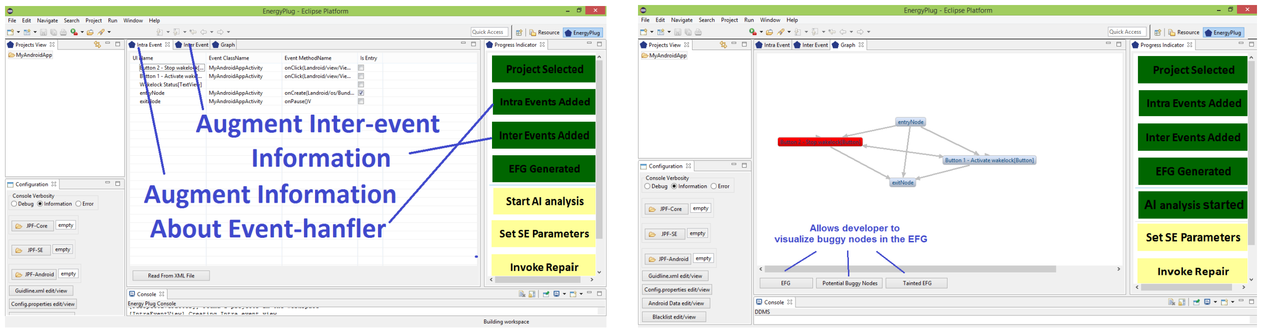

Validation
Driver Generator + Java Path Finder
1. Generates driver file
2. Executes symbolically
3. Generate test inputs (in case of assertion failure)
Repair
Repair code generation
1. Finds the class/method that needs to be fixed
2. Generates the repair expression
Dynodroid based EFG generation
(reads app class files & explores GUI automatically)
OR
XML Layout Parsing based EFG generation
(reads app class files & parses (GUI-related) XML files automatically. Edges between GUI events added manually)
App
(Eclipse workspace provides class files, XML layout, Manifest files, etc)
EFG
(Event Flow Graph)
Detection
Custom abstract interpretation engine
1. Analyses class files
2. Detects presence of energy bugs
3. Instruments class files for the validation phase
Set of system call APIs that are used to access energy heavy resources
Energy Bug Free / Repaired App

Fig. 16: Work flow inside EnergyPatch



(a) (b)

Fig. 19: Screenshot of the tool. (a) shows how developer can manually augment EFG information. (b) visualization inside tool showing information such as the structure of the EFG, buggy nodes, etc

by our framework. This global file is used to share the resource variable name and parameters to release system call from the resource acquiring activity to the exiting activity. This work-flow is also shown in Figure 17.

9 Eclipse Plugin EnergyPatch

Our framework has been implemented into an Eclipse plugin named EnergyPatch. The source code for the plugin is available under the BSD 3-Clause license [13] from BitBucket [3]. In this section we shall briefly describe the structure and working of the tool. An interested reader can find the instructions for installation and usage at http://www.comp.nus.edu.sg/~rpembed/epatch/home.html.

Figure 16 shows the work flow for EnergyPatch. In the plugin, the EFG generation can be done by using an automatic GUI exploration tool Dynodroid [8] or by parsing the GUI-related XML files from the app (In Android, it is common practice to specify the GUI layout of the app by means of XML files). We have also added an option (cf. Figure 19 (a)) that allows the developer to manually augment any additional flow dependencies (intra or inter event) within the EFG. This can be an useful feature in the cases where the automatic GUI exploration misses any event in the EFG during exploration.

For the detection phase, we have implemented an abstract interpretor for Java bytecode (for app class files). The abstract interpretation based analysis also requires the set of resources and associated system call APIs that need to be tracked with in the app. This information is also provided through an XML file and can be modified/replaced by the user as required. In case the abstract interpretor does not find any energy bugs, the analysis stops. However, in the case where potential energy bugs are found, the framework alerts the developer of same. This potential energy bug information is mapped to the EFG of analysed app and displayed in the graph view of the tool (Figure 19(b) shows an example). Such a pictorial representation may further assist the developer in understanding the debugging process.

In case a potential energy bug is found, the validation phase generates the driver code as described in Section 7.2. Also the required class files (from the analyzed app) are instrumented using ASM [14], as described in Section 6.5. Subsequently, the app (after search space reduction has been applied as described in Section 7.1) is executed symbolically. For Symbolic execution, our tool relies on Java Pathfinder (JPF), more specifically three components of JPF, JPF-core [15] , JPF-SE [16] and JPF-Android [17], are used. JPF-core provides the base JPF classes, JPF-SE provides the symbolic execution support and JPF-Android provides the model of Android framework. During symbolic execution, all assertion violations are reported to the developer. All reports are accompanied by a witness test-case that can be used to replicate the said energy bug using a mobile device and a power meter. Subsequently, the repair expressions are generated and presented to the developer as described in section 8.

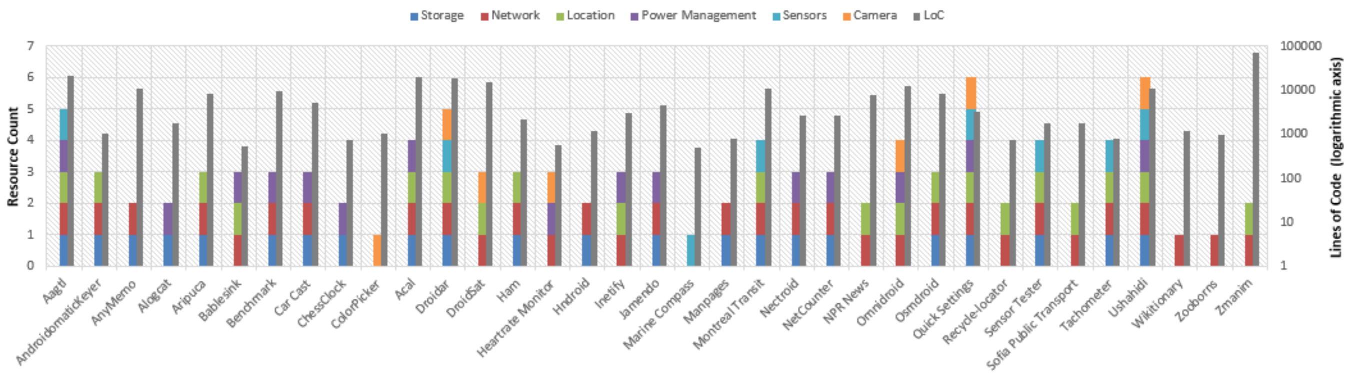


Fig. 20: Subject Apps; Resources used in the subject apps (Primary Y-axis) and Lines of Code (Secondary Y-axis, logarithmic)

TABLE 3: Subject apps for which energy bugs have been reported through bug-reports and/or previous publications [1], [18]–[20]

App Name version / code	App Description	LoC Apk Size(KB)	Event Handler Classes	Defect Description
Aripuca [21] 1.3.4 / 24 [22]	Recording tracks and saves waypoints	8093 660	14	Moving from MainActivity to WaypointActivity causes location updates to stay on even after the app is paused
□ Omnidroid 0.2.1 / 6 [23]	Automated event/action manager for Android	12425 258	28	Location updates started by the app are not stopped after app is paused or phone is restarted
Tachometer 1.0 / 1 [24]	App to measure location and speed	793 540	9	Selecting PositionActivity from main screen of the app causes location updates to be acquired but not released
△, □ Babblesink 1.0 / 1 [25]	An app to help locate lost phones	521 21	1	An exception may cause the app to have a potential wakelock bug
Sensor Tester 1.0 / 1 [26]	Sensor monitoring and logging app	1719 400	6	App acquires location and sensor services without releasing them on app pause
Aagtl [27] 1.0.31 / 31	Geocaching based app for Android	20572 307	4	Pressing Home button during cache download causes the wakelock to remain acquired by the app
□ DroidAR 1.0 / 1 [28]	An augmented-reality app for Android	18177 398	6	Going to the ArActivity then switching back to another activity cause the GPS to stay on even after closing app
Benchmark 1.1.5 / 9	Benchmarking app for Android	9739 1020	23	Navigating from Benchmark activity to show results causes the wakelock to be not released by the app
◇, □ Osmdroid 3.0.1 / 2 [29]	Provides replacement for Android's MapView	8107 276	10	Selecting the sample loader followed by first sample causes the app to not release the location updates
□ Recycle-locator 1.0 / 1 [30]	Area-specific restroom, mailbox finding app	717 116	3	Location services are not disabled when the map module is paused as a result GPS is constantly looking for a signal
□ SP Transport 1.17 / 18 [31]	Android app that assists in bus-travel	1766 161	3	Defective behaviour observed in the LocationView class, GPS is never turned off when the activity is paused
△, □ Ushaidi v2.2 / 13 [32]	App for Collection, visualization for crisis data	10621 713	22	CheckinMap keeps the GPS on, even after the user has navigated away from the activity
□ Zmanim 3.3.84.296 / 84 [33]	List of halachic/ halakhic times	72977 842	4	GPS signal acquisition from the ZmanimActivity is never stopped even after the app is paused

□: app used in [18]   △ : app used in [19]   ◇ : app used in [20]

10 Experimental Evaluation

Experiments for the evaluation of framework answer the following research questions: *(i)* Efficacy of our framework *i.e.* how effective is our framework in uncovering test cases that lead to real *energy bugs (ii)* Importance of the detection phase in our framework *(iii)* Effectiveness of the automated repair i.e. does the repair expression generated by our framework actually makes the application more energy efficient? and finally *(iv)* Comparison of our framework with existing works on resource leak detection in mobile apps. We shall discuss these research questions in Sections 10.2 - 10.5. However, first we shall discuss the experimental setup and choice of subject apps in Section 10.1.

10.1 Experimental Setup

Subject Apps: We created a suite of 35 subject apps from online sources such as F-droid app repository, Github, Google Code and Google Play, to be used in our experiments. These apps are diverse in terms of functionality, complexity and application size. In particular, the apps used in our experiments have at least 500 lines-of-code (LoC) and use one or more of energy-intensive resource(s). Figure 20 provides an overview of all the subject apps used in our experiments. These subject apps also include apps used in our previous works such as [1] and other related works such as [18], [19] and [20]. This will help us to compare our framework with existing related works. Table 3 lists down a few details of the subject apps for energy bugs were found/reported in previous works [1], [18]–[20]. These details include the app description, apk size and LoC, number of event-handlers and defect description for each of these apps. In Table 3, apps used in [18] are marked using the symbol □, apps used in [19] are marked using the symbol △ and apps from [20] are marked using the symbol ◇.

Setup: Our test-generation and repair framework was implemented in Java. It was run on a Desktop-PC with an Intel Core i7-2600 CPU with 8GB of RAM and Ubuntu 14.04 OS. The mobile device used to run the subject apps was an off-the-shelf LG Optimus E400 smartphone. The device has a 800 MHz Qualcomm processor, Adreno 200 GPU and features I/O components such as 3.2 inch TFT screen, GPS, Sensors, Wifi, 3G and Camera. This mobile device was running Android Gingerbread(v2.3.6) operating system(OS), which was the most widely used OS at the time of these experiments. It is worthwhile to know that newer versions of Android such as Android Jelly Bean and Ice Cream Sandwich have similar system call APIs (for resource usage) to that of Android Gingerbread (v2.3.6), therefore our framework should work equally well for apps intended for these platforms as well. Finally, for measuring energy savings in the patched apps we used a Yokogawa WT210 digital power meter using the a setup shown in Figure 22.

10.2 Efficacy of our Framework

The most important research question in the evaluation is determining the efficacy of our framework in finding and reporting energy bugs in real-life Android apps. To answer this we ran our framework for all subject apps (including the apps listed in Table 3) to observe, (i) if our framework could detect energy bugs and (ii) whether the test-cases generated our framework can be used to replicate these reported bugs on a real mobile-device.

In our experiments, our framework reported real energy bugs (with test-cases) for 12 out of these 35 apps. Among these 12 apps, 8 had energy bugs involving GPS (not all apps use the same APIs for accessing the GPS), 2 apps had energy bugs involving both the GPS and Sensors and 2 apps had energy bugs due to improper usage of Wakelocks. A summary of the key results can also be found in Table 4. *It is important to know that our framework reports the presence of energy bugs in an app only after both the detection (static analysis) and the validation (symbolic execution) phases have been completed.* It is also worthwhile to know that our

TABLE 4: Results of Detection/Validation phase for app listed in Table 3

App name	Resources Not Released	Detection Phase		Validation Phase	
		Time (s)	Scenario	Time (s)	No of Event-handlers Invoked in Test-case
Aripuca	GPS	21	Activity Switching	53	17
Omnidroid	GPS	3	Activity Switching	44	3
Tachometer	Sensor	<1	Resource Acquire Loop	12	6
	GPS				
Babblesink	Power Manager (Wakelock)	<1	Uncaught Exception	2	n/a (app crashes)
Sensor Tester	Sensor	4	Resource Acquire Loop	32	7
	GPS		Activity Switching		
Aagtl	Power Manager (Wakelock)	4	Activity Switching	28	4
DroidAR	GPS	6	Activity Switching	51	5
Benchmark	Power Manager (Wakelock)	4	Activity Switching	3	3
Osmdroid	GPS	5	Activity Switching	14	6
Recycle-locator	GPS	1	Activity Switching	3	4
SP Transport	GPS	2	Activity Switching	3	5
Ushaidi	GPS	4	Activity Switching	4	6
Zmanim	GPS	7	Activity Switching	34	5

framework has a relatively less performance overhead as both the computationally intensives phases *i.e.* the detection and validation phases, were completed in approximately a minute even though some of the application were significantly large with thousands of lines-of-code. This goes on to show that our framework can be applied to energy bug detection in real-life apps. On manual inspection of the apps for which potential energy bugs were detected, we observed following three scenarios:

i. *Activity Switching*: a resource is acquired in an activity, however the app navigates to another activity or stops execution without releasing the acquired resource.
ii. *Resource Acquire Loop*: a resource is repeatedly acquired within a loop however it is not released a sufficient number of times before exiting the application.
iii. *Uncaught Exception*: unexpected execution flow due to uncaught exceptions may leave resource(s) in the acquired state.

To check the usefulness of the generated test-cases we manually replayed these test-inputs on our test device and compared the resource status using the debugging tool Android debug bridge. By doing this additional step we were able to confirm that the test cases do indeed lead to buggy scenarios. Additionally, for some apps such as *DroidAR*, *Osmdroid*, *Recycle-locator*, *SP Transport* and *Ushaidi* there exists user reports (on code repositories) that describes the energy-related defect(s) to the developer. For these apps, we were able to compare the test cases generated by our framework to the test-scenarios reported/constructed by the user/developer. We observed significant similarities in these comparisons as well.

10.3 Importance of Detection Phase in the Framework

We have emphasized in previous sections (see Section 4), that the use of static analysis in the detection phase makes our framework scalable and also helps in reducing the overall analysis time. Here we shall present some observations to support these claims. We compare the (Symbolic Execution) SE only approach to the (Abstract Interpretation + Symbolic Execution) AI+SE approach for uncovering energy bugs, where

- *SE only* approach implies only symbolic execution is used to uncover energy bug(s) without the preceding static analysis
- *AI+SE* approach (our approach) implies that we perform static analysis (Abstract Interpretation or AI) followed by validation (Symbolic Execution or SE)

Specifically, we conducted experiments for two scenarios: *(i)* analysis time for both approaches in the absence of energy bugs, and *(ii)* analysis time for both approaches if energy bugs do exist. For the scenario where no energy bugs exist, static analysis terminates relatively fast (less than 15 seconds) when using the *AI+SE* approach (the one implemented in our framework). Additionally, since the results of the detection phase are always *sound*, we can be assured that no energy bug indeed exists at least for the portion of app represented by its EFG. However, to come to the same conclusion using *SE only* approach, all feasible program paths must be explored. Since there can be an unbounded number of event sequences in an app (because UI elements in an app can be navigated over a circular path), the SE only approach can potential take forever to conclude.

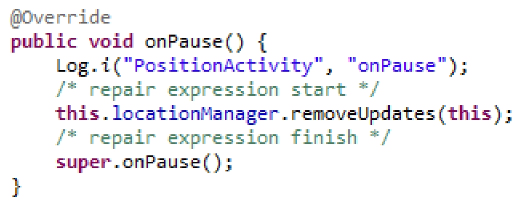
For the second scenario (where at least one energy bug exists in the analyzed app), the *AI+SE* approach can produce results in up to one-third of the time of *SE only* approach for certain apps. For instance, validation time for *Omnidroid [34]* was 117 seconds for SE only as compared to 44 seconds for the AI+SE approach, *cf* Table 5. This difference in evaluation time happens because the detection phase of our framework helps in search space reduction. The magnitude of search space reduction is directly influenced by the program location at which the (energy-bug-causing) resource/utility is acquired. The farther the (energy-bug-causing) program location is from the root UI node, the more the gains by using our search space reduction technique.

TABLE 5: Comparing execution time of SE-only approach Vs AI+SE (our) approach. Observed (geometric) mean improvement of 38.67%

App Name	Execution Time (s)		Improvement (%)
	SE-Only	AI+SE	SE-Only Vs AI+SE
Aripuca	109	53	51.3
Omnidroid	117	44	62.3
Tachometer	15	12	20.0
Sensor Tester	50	32	36.0
Aagtl	33	28	15.1
Benchmark	6	3	50.0
DroidAR	122	51	58.1
Osmdroid	17	14	17.6
Recycle-locator	5	3	40.0
SP Transport	7	3	57.1
Ushaidi	8	4	50.0
Zmanim	76	34	55.2

10.4 Effectiveness of Automated Repair

Our framework uses the test-cases generated by the validation phase to generate the repair expressions (described in Section 8). For instance, the test case for the app *Tachometer* that is shown in Figure 15, is used to generate the repair for class `PositionActivity.java` as shown in Figure 21.



```
@Override
public void onPause() {
    Log.i("PositionActivity", "onPause");
    /* repair expression start */
    this.locationManager.removeUpdates(this);
    /* repair expression finish */
    super.onPause();
}
```

Fig. 21: Repair expression for app Tachometer

To evaluate the effectiveness of the repair, we compared the energy consumption of the original app to that of the repaired app, for the buggy test-input. The setup for energy-measurement used a test device (LG Optimus E400 smartphone running Android v2.3.6) and a power meter (Yokogawa WT210) as shown in Figure 22. Energy-consumption of the device is measured for a period of 300 seconds *after* the bug revealing test-case has been executed. This is done to measure the impact of the buggy app code on the energy-consumption behaviour of the device. The test-cases were executed manually. Also no other apps were being executed on the test-device while the power-measurement experiments were being conducted. Power measurements were conducted thrice for each experiment and the average value for the readings were computed. Additionally, during these experiments the *screen timeout* duration of the device was set to 15 seconds. Table 6 shows the increase in energy-efficiency of the buggy apps before and after the repair code had been applied.

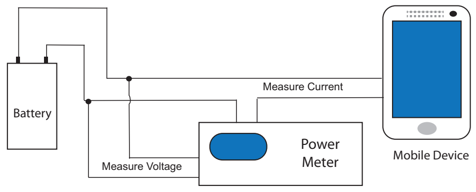


Fig. 22: Block diagram for measurement setup

TABLE 6: Improvement in energy consumption of all apps with validated energy bugs after the automatic repair

App Name	Energy Consumption (J)		Avg. Improvement
	Before Repair	After Repair	%
Aripuca	155.2	65.6	57.8
Omnidroid	71.2	66.4	6.7
Tachometer	159.5	66.7	58.2
Sensor Tester	155.8	66.5	57.3
Aagtl	68.3	64.4	5.7
Benchmark	91.1	64.2	29.5
DroidAR	156.9	66.5	57.6
Osmdroid	148.8	68.0	54.3
Recycle-locator	156.3	63.6	59.3
SP Transport	150.7	65.0	57.1
Ushaidi	154.8	65.3	57.8
Zmanim	158.1	65.4	58.7

10.5 Comparison with Existing Works on Resource Leak Detection in Mobile Apps

The works of [19], [20] and [18], are most related to our current work as they all present techniques for detecting and/or characterizing resource leaks in mobile apps. Therefore, in this subsection we shall discuss how our technique compares to these techniques in terms of (i) efficacy of defect detection and (ii) performance.

Efficacy of defect detection: The works of [19], [20] use static analysis to detect resource leaks in Android apps. In particular, [19] proposes a resource-leak detection technique based on data-flow analysis, whereas, the technique presented in [20] is based on function call graph traversal. Both of these works have observed that their techniques may produce some false positives. This is because unlike our framework, the techniques used in these works do not have a dynamic analysis or a test-generation phase, as a result there is no mechanism to generate test-cases or to automatically prune out the false positives that may be introduced due to the over-approximations of the static analysis phase. Unfortunately, we could only obtain the source-code for three apps from [19] and [20] (some of the apps from [20] were closed source whereas some programs used in [19] were individual class files from older versions of the Android framework). Those apps for which we could obtain the source-code, our framework was able to successfully find bugs in two of them (*Osmdroid* and *Ushaidi*) while for the third app *Babblesink* our framework reported that no feasible test-cases was present to trigger a resource leak. In particular for the *Babblesink* app, the detection phase of our framework reported a potential Wakelock related energy bug. However, analysis during the validation phase failed to produce a feasible test case that triggers the reported bug. Looking at the source code of the *Babblesink* app, we observed that a Wakelock object is acquired to ensure the interruption-free initialization of an *IntentService*. Ideally, this Wakelock should have been released after the initialization of the *IntentService* is completed. However, there exists a path in the application that will bypass the release instruction. This path is executed when an exception occurs after the acquire instruction for the Wakelock object has been executed but before the execution of the release instruction. This bug was initially reported by [19]. We manually inspected this app to find out the reason due to which the validation phase did not generate any valid test-case. We observed that the exception that caused the release code to be skipped actually crashes the app because it is uncaught up to the top level function of the application. Since all acquired Wakelocks are automatically released in the event of an app crash, no actual energy bug can occur in any feasible execution scenario. This is the reason why the validation phase in our framework (correctly) does not generate a feasible test-case for this app.

The work in [18] proposes a dynamic analysis based technique for resource leak detection. In particular, it uses bounded symbolic execution for finding test-cases that leads to resource leaks. In general, bounded symbolic execution implies that the depth at which symbolic exploration takes place is bounded. Bounding the depth of symbolic exploration may create limitations of its own. For instance, if in the example of Figure 8(f), the (loop) bound for exploration is set to 2 iterations, symbolic exploration would be unable to find any resource leaks. In general, knowing the adequate bound (such that all bugs can be revealed) can be quite challenging. Therefore, using only symbolic execution may not be optimal strategy for exploration. On the contrary, in our framework we first use static analysis to *conservatively*, detect the presence of resource leak, after which we use symbolic execution to generate test-cases. From the work of [18] we were able to obtain eight apps, seven out of which we were able to analyse successfully with our framework. For the eighth app, *Babblesink*, our framework did not produce test-cases for reasons mentioned in previous paragraph.

Performance: In this paragraph, we shall compare the performance of [19], [20] and [18] with our technique. We observed that the static analysis based techniques that were described in the works of [19] and [20] have similar performance to that of the static analysis based detection phase that was used in our framework. In particular, the results reported in [19] stated a minimum analysis time of 3 seconds (for 0.3KLoC) to a maximum analysis time of 408 Seconds (for 93.5KLoC). Whereas, the results reported in [20] stated a minimum analysis time of 6 seconds (for 345 Classes, 1447 Methods) to a maximum analysis time of 61 seconds (for 420 Classes, 3804 Methods) (LoC is not reported for the apps used in [20]). In contrast, the performance of the bounded symbolic exploration based dynamic exploration technique used in [18], heavily depends on the depth of exploration. This means that the bound set on bounded symbolic exploration technique will influence the number of symbolic states that need to be explored and therefore the performance of the technique itself. In particular, the results reported in [18] stated that when the bound (for this particular technique, bound is number of user interactions) was set to 6, the analysis time varied between 11 seconds (for 0.8KLoC) to 284 seconds (for 18KLoC). However, when the exploration bound is increased to 8, the technique of [18] takes over an hour to analyze several of the subject applications. Such a result is intuitive as exploration of more symbolic states requires more execution time. In comparison, our framework uses a static analysis based detection phase to detect potential energy bugs, whereas the use of symbolic exploration in our framework is only to validate the potential energy bugs by executing small portions of the app. As a result the validation phase in our framework is able to terminate in a relatively short amount of time (observed timings less than a minute, see Table 3).

11 Threats to Validity

One of the major threats to the validity of the results produced by our framework is due to the incompleteness of the EFGs used in our analysis. As described in Section 5, the EFG for an app is generated using a dynamic analysis technique. Since we cannot guarantee the completeness of the EFG, therefore, we cannot provide any completeness guarantee for the generated results as well. As a consequence of this limitation, in case of incomplete EFG our framework may leave portions of the app code unanalyzed that have not been represented in the EFG.

Another threat to the validity of the results produced by our framework is due to the use of Android framework model. In our framework we use an Android framework model in order to ensure the correct execution of an app. Our Android framework model is based on an existing model proposed in [17]. It is worthwhile to mention that defects in the Android framework model can affect the results of our framework, therefore we invest additional effort to ensure that the model closely mimics the behaviour of the Android framework.

12 Related Work

Techniques based on profiling: Recent profiling based works [35]–[37] have demonstrated the presence of energy-inefficiencies in several popular real-life mobile applications. The key idea in such works is to execute the application for several test inputs, while monitoring the energy consumption of the mobile device. One of these works, Eprof [35], for instance instruments the application code such that all invocations of system-call APIs can be recorded during profiling. Instrumented apps are then run on a real mobile device and invocation of system call APIs are recorded. These recorded system-call API invocation traces are then used in combination with a FSM power model of the mobile device to estimate the energy-consumption behaviour of the app. Subsequent works, such as eLens [36] and vLens [37] have presented more fine-grained profiling techniques for mobile apps. In particular, eLens and vLens provide an estimated energy-consumption for each line of source-code in an app. This is unlike Eprof which provides the energy-consumption estimation for an app on a system-call API level of abstraction. A key difference between the works of eLens and vLens is that the technique of eLens uses an power-model (based on the work of [38]) to provide a per-line energy-estimation of an app whereas the technique of vLens uses actual power measurements to provide the same information. It is worthwhile to know that like any other profiling techniques, the techniques mentioned in this paragraph are heavily dependent on test inputs to obtain the profile. Obtaining (or randomly guessing) test inputs that lead to energy bugs would be difficult, in the general case. However, we feel that such profiling based techniques can benefit from our work as they can use the test cases generated by EnergyPatch to obtain the energy profile of the application under test. In addition, some of these works [35], [36] require power-models in order to generate energy-consumption estimates. Therefore, in the following paragraph we discuss some of the existing works that have been proposed for the purposes of power-model generation.

Techniques for power-modeling: [39] presents one of the earliest works on the topic of instruction level power-modeling. Other works such as [38] have proposed the use of dynamic analysis technique to generate power-models. Essentially such techniques present mechanisms to associates each instruction with an energy consumption cost. The technique of [38] in particular, has been extended to work for mobile-devices in the work of [36]. One obvious complexity with such techniques is that the power-model generated by these works is hardware-specific, therefore it must be re-generated every time the hardware changes. Additional complexities may arise (in creating the power-model), due to program-specific behaviour such as cache misses, branch mis-predictions, etc. A number of subsequent works, such as [40]–[42], propose techniques for automated or semi-automated means of power-model generation and may potentially reduce the complexity of developing power-models. The work of [40] proposes a tool PowerBooter for constructing power-model for a mobile device using built-in battery voltage sensors. The work of [41] has an objective similar to that of the work presented in [40], however the power-model generated using the technique of [41] is capable of producing a much finer-grained power-model than [40]. The work of [42] presents a tool WattsOn that can be used for emulating energy-consumption characteristics of a mobile-device. Such automated power-model generation techniques may be quite useful for profiling-based technique presented in previous paragraph, however, the objective of our work is quite different from such power-model based works. In our work we are primarily interested in finding scenarios (test-cases) that lead to energy-inefficient behaviour such as energy bugs, instead of finding the energy-consumption behaviour of the entire app.

Techniques for detecting resource leaks, memory leaks: There exists a number of works that present techniques for detecting memory leaks [43]–[49], as well as resource leaks (such as file handlers, semaphores, *etc*) [50], [51]. These works address different aspects of detecting leaks in programs. For instance, the work of [43] proposes a sampling based technique that specifically targets memory-leaks that are present in the in-frequently executed portions of the program source-code. Another profiling work [46] proposes a memory leak detection technique specifically targeted at object-oriented programs. Static analysis has also been used to detect memory leaks [44], [45] and resource leaks [50]. Combination of static and dynamic analysis techniques have also been used to detect memory leaks in the work of [49]. Several other approaches have been proposed for the purpose of leak detection. For instance, [48] proposes a learning based technique, [47] proposes a tainting based technique and [51] proposes a resource usage pattern based leak detection technique.
It is worthwhile to know that memory leaks and resource leaks affect the behaviour of a program in different ways than energy bugs. Presence of memory leaks can be detrimental to the performance of program (adverse for time-critical applications); presence of resource leaks may cause deadlocks or delays that may influence the functionality of the program. Whereas the presence of energy bugs does not affect the execution of the program in any way. Instead the presence of energy bugs in mobile applications indirectly affects the battery life of the mobile device and the user experience (by reducing the amount of time the mobile device can stay functional). Hence, the problems posed by energy bugs have different characteristics to that of memory/resource leaks and should be studied separately.

Techniques using program analysis to detect energy-inefficiencies: Existing works have proposed a number of complimentary ways to make apps more energy-efficient. For instance the work of [52] proposes a technique to identify resource-intensive pieces of code at runtime and execute them at remote servers. This provides an indirect way of increasing the battery life of the mobile device. Another category of work exists that uses program analysis techniques to uncover energy-inefficient behaviour in apps so that the developer can identify and fix problem with the app before deployment. In this paragraph we shall primarily discuss the works in this category as it is more relevant to our current work. Recent works such as [19], [20] propose static analysis based techniques for detecting resource leaks in Android apps. The work of [19] in particular is one of the earliest works in this domain however the technique proposed by this work does not focus on systematic representation and exploration of event-sequences in the app. As a result, some manual effort may be required to generate and validate energy-inefficiencies that have been detected by this technique. In addition, since such techniques solely rely on static analysis and so their results may contain false positives (in the presence of infeasible paths). There exists a number of research works, such as [38], [53], that have proposed dynamic analysis based techniques for energy-inefficient behaviour detection in mobile-apps. The work of [38] in particular uses symbolic execution to explore paths in an app and subsequently highlight the CPU energy consumption for each path. Whereas the work of [53] explores states in the GUI model of an app to detect resource leaks. There are a number of potential limitations to such techniques. For instance the technique of [38] only considers the energy consumption due to the CPU, however most modern mobile devices have a number of hardware components that have power consumption comparable to that of the CPU. A more serious drawback with such techniques is that they rely on exploring all paths in the program, which may not be scalable for many real-life programs.

Techniques for test generation in mobile-apps: Works related to test generation in mobile apps have mostly been confined to the domain of functionality testing. Testing mobile application such as Android apps can be challenging as they do not have common entry point (or *main* method), unlike Java programs. Instead such apps are composed of several disconnected pieces of code called event-handlers that are responsible for handing events (such as user taps or touches). As a result, automated exploration of these apps is usually non-trivial. Research works such as [8], [54] have proposed a number of automated techniques to explore mobile-apps that can be used for functionality testing purposes. [8] in particular uses a biased-random testing technique to explore the various GUI states of an app. The technique proposed in [54] uses symbolic execution to achieve the same goal. It is worthwhile to know that even though these techniques by themselves cannot be used to test non-functional properties like energy-consumption but they do provide insights that can be useful for developing automated non-functionality testing tools for mobile-apps.

Techniques for energy-aware test generation mobile-apps: At the time of writing, there were two different approaches, [1] and [18], on the topic of energy-aware test-generation for mobile-apps. One of the approaches is our previous work [1] which uses a hardware-software hybrid approach to systematically generate test inputs that leads to energy-inefficient scenarios. As the section 3.3 mentions, the technique of [1] differs from our current work in three key aspects (i) the way energy bugs are detected (power measurement vs static analysis) (ii) the way test-cases are generated (search heuristics vs guided symbolic execution) and (iii.) automatic repair expressions generation (present only in the current framework).

The other related work, Greendroid [18], has a number of conceptual and technical differences from our current work. The technique in Greendroid primarily relies on bounded symbolic execution to detect resource leaks in mobile apps. As described in section 10.5, bounded symbolic execution may be unable to detect feasible resource leaks, if the bounds (for the bounded explorations) are set too cautiously. In contrast, our framework uses static analysis to *conservatively*, detect the presence of resource leaks in mobile apps. In our framework, symbolic execution is used to validate the presence of these potential resource leaks and to generate test-cases. In addition, our framework is also capable of generating repair expression automatically, whereas the repair part in Greendroid requires manual effort.

Another more subtle technical difference between our current work and Greendroid is the granularity at which symbolic exploration is conducted. Greendroid defines inputs as a sequence of events. However, as demonstrated by means of an example in section 2.2, such a definition may lead to insufficient exploration (of program paths). This is because execution of certain program-paths inside the event handler may depend on the return value of Android system call APIs (such as the path $N1 - N2 - N4\dots$ in event handler $E3$ in Figure 9, depends on the return value of syscall1()). In our framework, we define an input as a combination of events (either user-generated or system-generated) and return value from system call APIs (*cf.* Definition 1). Our framework can not only explore the event-handlers of an app but also the program-paths inside these event-handlers. For example, in the event handler $E3$ of Figure 9, the return values of system calls syscall1() and syscall2() are made symbolic, such that paths inside event handler $E3$ can be explored.

Techniques for green software engineering: Another line of work [55]–[58] exists that does not rely on energy-aware test-generation to optimize energy-efficiency of programs. The work of [55] for instance introduces a new program construct (energy types) that allows the developer to annotate energy-consumption information within the app source-code. This annotated energy-consumption information, along with power-management features such as dynamic voltage & frequency scaling, can be used to develop software-hardware interaction based energy-management techniques. Another set of work [56], [57] focuses on optimizing energy-consumption behaviour of a program by utilizing different implementations of same functionality. The different implementation for a given functionality can be obtained semi-automatically [56] or automatically [57]. For instance, the work of [56] provides program constructs that can be used by the developer to indicate energy-intensive functionalities (such as functions, loops) and to provide alternative implementations for these energy-intensive functionalities. The decision to choose an implementation is influenced by monitoring the runtime energy-consumption behaviour of the program for a given workload. Along the same lines, the work of [57] proposes a framework that can optimize energy-efficiency of Java applications by choosing the most energy-efficient library implementation for Java Collection API. Finally, recent works such as [58] have proposed automated techniques to reduce energy-efficiency of program by targeting specific energy-intensive functionalities such as sending HTTP requests.

13 Conclusion

In this paper, we presented a framework that can provide an end-to-end solution for detecting, validating and repairing energy bugs in real-life mobile apps. The use of light-weight static analysis technique in the detection phase allows us to quickly narrow down the potential program paths along which energy bugs may occur. Subsequent exploration of these potentially-buggy program paths using dynamic analysis technique helps us in validating these potential energy bugs. Our framework also generates test-cases for all validated energy-bugs which can be used by the app-developer to manually recreate the buggy scenarios. Finally, our framework generates repair expression to fix the validated energy bugs. We implemented our framework as an Eclipse plugin so that it can be easily installed and used by app-developers during app-development. We conducted experiments to evaluate the effectiveness and efficacy of our framework by testing real-life Android apps. During these experiments our framework reported real energy bugs in twelve out of thirty-five tested apps. Also the test-cases generated by our framework were able to trigger the reported energy bugs on a real mobile-device. Finally, we compared the energy-consumption of the buggy apps post repair on our test-device. In these experiments we observed that the repair code generated by our framework can improve the energy-efficiency of the buggy apps significantly.

Acknowledgement

The work was partially supported by a Singapore MoE Tier 2 grant MOE2013-T2-1-115 “Energy aware programming”.

References

[1] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, “Detecting energy bugs and hotspots in mobile apps,” in *FSE*, 2014.
[2] “GSM Arena,” http://www.gsmarena.com/lg_optimus_l3_e400-4461.php.
[3] “EnergyPatch,” https://bitbucket.org/abhijeet_code/energypatch.
[4] “Android develeoper website, wifimanager,” http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html.
[5] “Android develeoper website, powermanager,” http://developer.android.com/reference/android/os/PowerManager.html.
[6] “Android develeoper website, sensormanager,” http://developer.android.com/reference/android/hardware/SensorManager.html.
[7] A. Pathak, Y. C. Hu, and M. Zhang, “Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices,” in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*.
[8] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: an input generation system for android apps,” in *ESEC/SIGSOFT FSE*, 2013.
[9] A. M. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: Reverse engineering of graphical user interfaces for testing,” in *WCRE*, 2003.
[10] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *POPL*, 1977.
[11] “Android service lifecycle,” https://developer.android.com/guide/components/services.html.
[12] S. Anand, C. S. Păsăreanu, and W. Visser, “JPF-SE: A symbolic execution extension to java pathfinder,” in *TACAS*, 2007.
[13] “Bsd-3-clause license,” http://opensource.org/licenses/BSD-3-Clause.
[14] “Asm, java bytecode manipulation and analysis framework,” http://asm.ow2.org/.
[15] K. Havelund and T. Pressburger, “Model checking java programs using java pathfinder,” *International Journal on Software Tools for Technology Transfer*, 2000.
[16] S. Anand, C. S. Păsăreanu, and W. Visser, “JPF-SE: A symbolic execution extension to java pathfinder,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007.
[17] H. van der Merwe, B. van der Merwe, and W. Visser, “Verifying android applications using java pathfinder,” *ACM SIGSOFT Software Engineering Notes*, 2012.
[18] Y. Liu, C. Xu, S. Cheung, and J. Lu, “Greendroid: Automated diagnosis of energy inefficiency for smartphone applications,” *Software Engineering, IEEE Transactions on*, 2014.
[19] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, “What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps,” in *MobiSys*, 2012.
[20] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, “Characterizing and detecting resource leaks in android applications,” in *ASE*, 2013.
[21] “Aripuca,” https://play.google.com/store/apps/details?id=com.aripuca.tracker&hl=en.
[22] “Aripuca gps tracker,” https://f-droid.org/wiki/index.php?title=com.aripuca.tracker.
[23] “Omnidroid,” https://f-droid.org/wiki/page/edu.nyu.cs.omnidroid.app.
[24] “Speedometer,” https://play.google.com/store/apps/details?id=com.bjcreative.tachometer.
[25] “Bablesink,” https://github.com/hatstand/babblesink/.
[26] “Sensor Tester,” https://play.google.com/store/apps/details?id=com.dicotomica.sensortester.
[27] “Aagtl,” https://play.google.com/store/apps/details?id=com.zoffcc.applications.aagtl.
[28] “Droidar, issue 27,” https://code.google.com/p/droidar/issues/detail?id=27.
[29] “Osmdroid, issue 53,” https://code.google.com/p/osmdroid/issues/detail?id=53.
[30] “Recycle-locator, issue 33,” https://code.google.com/p/recycle-locator/issues/detail?id=33.

[31] "Sofia public trasport, issue 76," https://github.com/ptanov/sofia-public-transport-navigator/issues/76.

[32] "Ushahidi, issue 11," https://github.com/ushahidi/Ushahidi_Android/pull/11.

[33] "Halachic prayer times," https://f-droid.org/wiki/page/net.sf.times.

[34] "Omnidroid," https://f-droid.org/repository/browse/?fdid=edu.nyu.cs.omnidroid.app.

[35] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *EuroSys*, 2012.

[36] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *ICSE*, 2013.

[37] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *ISSTA*, 2013.

[38] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat, "SEEP: exploiting symbolic execution for energy-aware programming," ser. HotPower, 2011.

[39] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," in *ICCAD*, 1994.

[40] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, ser. CODES/ISSS '10, 2010.

[41] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11, 2011.

[42] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, ser. Mobicom '12, 2012.

[43] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI, 2004.

[44] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005.

[45] M. Orlovich and R. Rugina, "Memory leak analysis by contradiction," in *Proceedings of the 13th International Conference on Static Analysis*, ser. SAS'06, 2006.

[46] D. Rayside and L. Mendel, "Object ownership profiling: A technique for finding and fixing memory leaks," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007.

[47] J. Clause and A. Orso, "Leakpoint: Pinpointing the causes of memory leaks," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, 2010.

[48] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, 2010.

[49] M. Li, Y. Chen, L. Wang, and G. Xu, "Dynamically validating static memory leak warnings," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013, 2013.

[50] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, 2010.

[51] H. Zhang, G. Wu, K. Chow, Z. Yu, and X. Xing, "Detecting resource leaks through dynamical mining of resource usage patterns," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, ser. DSNW '11, 2011.

[52] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10, 2010.

[53] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in android applications," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, 2013.

[54] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *SIGSOFT FSE*, 2012.

[55] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu, "Energy types," in *OOPSLA*, 2012.

[56] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

[57] I. Manotas, L. Pollock, and J. Clause, "SEEDS: A software engineer's energy-optimization decision support framework," in *ICSE*. ACM/IEEE, 2014.

[58] D. Li, Y. Lyu, J. Gui, and W. G. Halfond, "Automated energy optimization of http requests for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, May 2016.