ISR
Institute for Software Research
University of California, Irvine

An Energy-Aware Mutation Testing Framework for Android

Reyhaneh Jabbarvand
University of California, Irvine
jabbarvr@uci.edu

Sam Malek
University of California, Irvine
malek@uci.edu

January 2017
ISR Technical Report # UCI-ISR-17-1

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
isr.uci.edu

An Energy-Aware Mutation Testing Framework for Android

Reyhaneh Jabbarvand and Sam Malek
Institute of Software Research, University of California, Irvine
{jabbarvr,malek}@uci.edu

***Abstract*—The rising popularity of mobile apps deployed on battery-constrained devices underlines the need for effectively evaluating their energy properties. However, currently there is a lack of testing tools for evaluating the energy properties of apps. As a result, for energy testing, developers are relying on tests intended for evaluating the functional correctness of apps. Such tests may not be adequate for revealing energy defects and inefficiencies in apps. This paper presents an energy-aware mutation testing framework, called *μDroid*, that can be used by developers to assess the adequacy of their test suite for revealing energy-related defects. *μDroid* implements fifty energy-aware mutation operators and relies on a novel, automatic oracle to determine if a mutant can be killed by a test. Our evaluation on real-world Android apps shows the ability of proposed mutation operators for evaluating the utility of tests in revealing energy defects. Moreover, our automated oracle can detect whether tests kill the energy mutants with an overall accuracy of 94.6%, thereby making it possible to apply *μDroid* automatically.** [1]

I. Introduction

Energy is a demanding but limited resource on mobile and wearable devices. Improper usage of energy consuming hardware components, such as GPS, WiFi, Radio, Bluetooth, and display, can drastically discharge the battery. Recent studies have shown energy to be a major concern for both users [51] and developers [45]. In spite of that, many mobile apps are abound with energy defects.

The majority of apps are developed by start-up companies and individual developers that lack the resources to properly test their programs. The resources they have are typically spent on testing the functional aspects of apps. However, tests designed for testing functional correctness of a program may not be suitable for revealing energy defects. In fact, even in settings where developers have the resources to test the energy properties of their apps, there is generally a lack of tools and methodologies for energy testing [45]. Thus, there is an increasing demand for solutions that can assist the developers in identifying and removing energy defects from apps prior to their release.

One step toward this goal is to help the developers with evaluating the quality of their tests for revealing energy defects. *Mutation testing* is an approach for evaluating fault detection ability of a test suite by seeding the program under test with artificial defects, a.k.a, *mutation operators* [32], [35]. Mutation operators can be designed based on a defect model, where mutation operators create instances of known defects, or by mutating the syntactic elements of the programming language. The latter creates enormously large number of mutants and makes energy-aware mutation testing infeasible, as energy testing should be performed on a real device to obtain accurate measurements of battery discharge. Additionally, energy defects tend to be complex (e.g., manifest themselves through special user interactions or peculiar sequence of external events). As Rene et al. [36] showed complex faults are not highly coupled to syntactic mutants, energy-aware mutation operators should be designed based on a defect model.

In this paper, we present *μDroid*, an energy-aware mutation testing framework for Android. We have focused our effort on Android, as it is the most widely used mobile platform. However, we believe the concepts underlying the framework are applicable to other mobile platforms as well. In the design of *μDroid*, we had to overcome two challenges:

1) An effective approach for energy-aware mutation testing needs an extensive list of *energy anti-patterns* in Android to guide the development of mutation operators. An energy anti-pattern is a commonly encountered development practice (e.g., misuse of Android API) that results in unnecessary energy inefficiencies. While a few energy anti-patterns, such as resource leakage and sub-optimal binding [42], [43], [52], have been documented in the literature, they do not cover the entire spectrum of energy defects that arise in practice. To that end, we first conducted a systematic study of various sources of information, which allowed us to construct the most comprehensive energy defect model for Android to date. Using this defect model, we designed and implemented a total of *fifty* mutation operators that can be applied automatically to apps under test.

2) An important challenge with mutation testing is the *oracle problem*, i.e., determining whether the execution of a test case kills the mutants or not. This is particularly a challenge with energy testing, since the state-of-the-practice is mostly a manual process, where the engineer examines the power trace of running a test to determine the energy inefficiencies that might lead to finding defects. To address this challenge, we present a novel, and fully automated oracle that is capable of determining whether an energy mutant is killed by comparing the power traces of tests executed on the original and mutant versions of an app.

We have evaluated *μDroid* using open-source Android apps and in terms of its ability to evaluate quality of tests for energy

[1] ISR Technical Report UCI-ISR-17-1, January 2017

testing, helping the developers create better tests, the accuracy of its oracle, and its performance. Our experiments show that *μDroid* can effectively and efficiently evaluate the adequacy of test suites for energy testing, leveraging an automated oracle with an average accuracy of 94.6%.

This paper makes the following contributions:

- Comprehensive list of energy anti-patterns collected from issue trackers, Android developers guide, and Android API reference.
- Design of fifty energy-aware mutation operators based on the energy anti-patterns and their implementation as an Eclipse plugin, which is publicly available [10].
- A novel automatic oracle for mutation analysis to identify if an energy mutant can be killed by a test suite.

The remainder of this paper is organized as follows. Section II provides an overview of our framework. Sections III describes our extensive study to collect energy anti-patterns from variety of sources and presents the details of our mutation operators with several coding examples. Section IV introduces our automated approach for energy-aware mutation analysis. Section V presents the implementation and evaluation of the research. Finally, the paper outlines related research and concludes with a discussion of our future work.

II. Framework Overview

Figure 1 depicts our framework, *μDroid*, for energy-aware mutation testing of Android apps, consisting of three major components: (1) *Eclipse Plugin* that implements the mutation operators and creates a mutant from the original app; (2) *Runner/Profiler* component that runs the test suite over both the mutated and original versions of the program, profiles the power consumption of the device during execution of tests, and generates the corresponding power traces (i.e., time series of profiled power values); and (3) *Analysis Engine* that compares the power traces of tests in the original and mutated versions to determine if a mutant can be killed by tests or not.

Our Eclipse plugin implements *fifty* energy-aware mutation operators (details in Section III) derived from an extensive list of energy anti-patterns in Android. To generate mutants, our plugin takes the source code of an app and extracts the Abstract Syntax Tree (AST) representation of it. It then searches for anti-patterns encoded by mutation operators in AST, transforms the AST according to the anti-patterns, and generates the implementation of the mutants from the revised AST.

After generating a mutant, the Runner/Profiler component runs the test suite over the original and mutant versions, while profiling the actual power consumption of the device during execution of test cases. This component creates the power trace for each test case that is then fed to the Analysis Engine.

Analysis engine employs a novel algorithm to decide whether each mutant is killed or lived. At a high-level, it measures the similarity between time series generated by each test after execution on the original and mutated versions of an app (details in Section IV). In doing so, it accounts for distortions in the collected data. If the temporal sequences of power values for a test executed on the original and mutated app are not similar, Analysis Engine marks the test as killed. A mutant lives if none of the tests in the test suite can kill it.

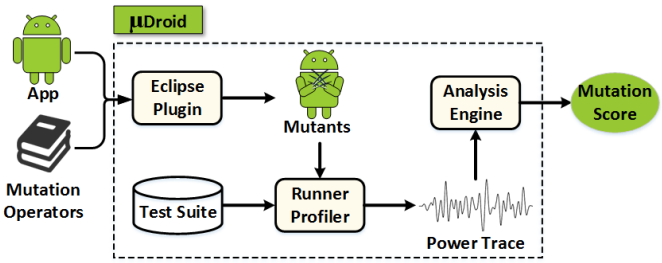

Fig. 1: Energy-aware mutation testing framework

The implementation of our framework is extensible to allow for inclusion of new mutation operators, Android devices, and analysis algorithms. In the following two sections, we describe the details of our energy-aware mutation operators and mutation analysis engine.

III. Energy-Aware Mutation Operators

To design the mutation operators, we first conducted an extensive study to identify the commonly encountered energy defects in Android apps, which we call *energy anti-patterns*. To that end, we explored bug repositories of open-source projects, documents from Google and others describing best practices of avoiding energy inefficiencies, and published literature in the area of green software engineering.

A. Defect Model and Derivation of the Mutation Operators

Our methodology to collect the energy anti-patterns was a *keyword-based* search approach. We started by crawling Android Open Source Project issue tracker [3] and XDA Developers forum [18] and searched for posts that have at least one of the following keywords: *energy*, *power*, *battery*, *drain*, and *consumption*. The outcome of this step was a list of energy related issues, and *295* apps where those issues had been reported. We excluded commercial apps from the list, since our research requires the availability of source code. That left us with *130* open-source apps for further investigation.

We then searched the issue tracker of the *130* apps for the aforementioned keywords, and narrowed down to *91* open source apps that had at least one issue (open or closed) related to energy, as reported in their issue tracker. We considered apps whose energy issues were reproducible by the developers—whether confirmed or fixed—or had an explanation as to how to reproduce the issue, which left us with *41* apps. Moreover, we studied the related literature [20], [21], [30], [42], [43], [52] and found *18* additional open-source apps with energy issues. In the end, we were able to identify *59* open-source apps with confirmed energy defects.

We manually investigated the source code of these *59* apps to find misuse of Android APIs utilizing energy-expensive hardware components (e.g., CPU, WiFi, Radio, Display, GPS, Bluetooth, and Sensors) as reported in the corresponding bug trackers. For example, Omim [9] issue

TABLE I: List of proposed energy-aware mutation operators.

Category	Description of Classes	Mutation Operators	Hardware	Type
Location	Increase Location Update Frequency	LUF_T, LUF_D	GPS/WiFi/Radio	R
	Change Location Request Provider	LRP_C, LRP_A	GPS/WiFi/Radio	R,I
	Redundant Location Update	RLU, RLU_D, RLU_P	GPS/WiFi/Radio	D
	Voiding Last Known Location	LKL	GPS/WiFi/Radio	R
Wakelock	Wakelock Release Deletion for CPU	WRDC, WRDC_P, WRDC_D	CPU	D
	Keep WakefulBroadcastReceiver Active	WBR	CPU	D
	Wakelock Release Deletion for WiFi	WRDW, WRDW_P, WRDW_D	WiFi	D
	Acquire High Performance WiFi Wakelock	HPW	WiFi	R
Connectivity	Fail to Check for Connectivity	FCC_A, FCC_R	WiFi/Radio	R,I
	Frequently Scan for WiFi	FSW_H, FSW_S	WiFi	R
	Redundant WiFi Scan	RWS	WiFi	D
	Use Cellular over WiFi	UCW_C, UCW_W	WiFi/Radio	I
	Long Timeout for Corrupted Connection	LTC	WiFi/Radio	R,I
	Downloading Redundant Data	DRD	WiFi/Radio	D
	Unnecessary Active Bluetooth	UAB	Bluetooth	R
	Frequently Discover Bluetooth Devices	FDB_H, FDB_S	Bluetooth	R
	Redundant Bluetooth Discovery	RBD	Bluetooth	D
Display	Enable Maximum Screen Timeout	MST	Display	I
	Set Screen Flags	SSF	Display	I
	Use Light Background Color	LBC	Display	R
	Enable Maximum Screen Brightness	MSB	Display	I
Recurring Callback and Loop	High Frequency Recurring Callback	HFC_T, HFC_S, HFC_A, HFC_H	WiFi/Radio/CPU/Memory/Bluetooth/Display	R
	Redundant Recurring Callback	RRC	WiFi/Radio/CPU/Memory/Bluetooth/Display	D
	Running an Alarm Forever	RAF	WiFi/Radio/CPU/Memory/Bluetooth/Display	D
	Battery-Related Frequency Adjustment	BFA_T_F, BFA_T_L, BFA_S_F, BFA_S_L, BFA_A_F, BFA_A_L, BFA_H_F, BFA_H_L	WiFi/Radio/CPU/ Memory/ Bluetooth/ Display	I
	Increasing Loop Iterations	ILI	WiFi/Radio/CPU/Memory/Bluetooth/Display	I
Sensor	Sensor Listener Unregister Deletion	SLUD	Sensors	D
	Fast Delivery Sensor Listener	FDSL	Sensors	R

780 states *"App is using GPS all the time, or at least trying to use"*. As a result, we investigated usage of APIs belonging to `LocationManager` package in Android. As another example, SipDroid [15] issue 847 states *"after using the app, display brightness is increased almost full and its stays that way"*. Thereby, we investigated the source code for APIs dealing with the adjustment of screen brightness, e.g., `getWindow().addFlag(FLAG_KEEP_SCREEN_ON)` and `getWindow().getAttributes().screenBrightness`.

In addition to bug trackers, we crawled Android Developers Guide [2] and Android API Reference [1] for best practices related to energy consumption using the aforementioned keywords. We identified 28 types of energy anti-patterns from our investigation, which were used to design our energy-aware mutation operators for the purpose of this work.

Table I lists our energy-aware mutation operators. We designed and implemented 50 mutation operators (column 3 in Table I)—corresponding to the identified energy defect patterns, grouped into 28 classes (column 2 in Table I). We also categorized these classes of mutation operators into 6 categories, which further capture the commonality among the different classes of operators. Each row of the table presents one class of mutation operators, providing (1) a brief description of the operators in the class, (2) the ID of mutation operators that belong to the class, (3) list of the hardware components that the mutation operators might engage, and (4) modification types made by the operators (R: Replacement, I: Insertion, D: Deletion).

Due to space constraints, in the following sections, we describe a subset of our mutation operators. Details about all mutation operators can be found on the project website [10].

B. Location Mutation Operators

When developing location-aware apps, developers should use a location update strategy that achieves the proper tradeoff between accuracy and energy consumption [7]. User location can be obtained by registering a `LocationListener`, implementing several callbacks, and then calling `requestLocationUpdates` method of `LocationManager` to receive location updates. When the app no longer requires the location information, it needs to stop listening to updates and preserve battery by calling `removeUpdates` of `LocationManager`. Though seemingly simple, working with Android `LocationManager` APIs could be challenging for developers and cause serious energy defects.

Figure 2 shows a code snippet inspired by real-world apps that employs this type of API. When `TrackActivity` is launched, it acquires a reference to `LocationManager` (line 5), creates a location listener (lines 6-10), and registers the listener to request location updates from available providers every 2 minutes (i.e., $2*60*1000$) or every 20 meters change in location (line 11). Listening for location updates continues until the `TrackActivity` is destroyed and the listener is unregistered (line 16).

We provide multiple mutation operators that manipulate the usage of `LocationManager` APIs. LUF operators increase the frequency of location updates by replacing the second (LUF_T) or third (LUF_D) parameters of `requestLocationUpdates` method with 0, such that the app requests location notifications more frequently.

If LUF mutant is killed (more details in Section IV), it shows the presence of at least one test in the test suite

```
public class TrackActivity extends Activity {
  private LocationManager manager;
  private LocationListener listener;
  protected void onCreate(){
    manager = getSystemService("LOCATION_SERVICE");
    listener = new LocationListener(){
      public void onLocationChanged(){
        // Use location information to update activity
      }
    };
    manager.requestLocationUpdates("NETWORK", 2*60*1000,
        20, listener);
  }
  protected void onPause(){super.onPause();}
  protected void onDestroy(){
    super.onDestroy();
    manager.removeUpdates(listener);
  }
}
```

Fig. 2: Example of obtaining user location in Android

```
protected void downloadFiles(String link){
  WifiLock lock = getSystemService().createWifiLock();
  lock.acquire();
  URL url = new URL(link);
  ConnectivityManager manager = getSystemService(
      "CONNECTIVITY_SERVICE");
  NetworkInfo nets = manager.getActiveNetworkInfo();
  if(nets.isConnected()){
    HttpURLConnection conn = url.openConnection();
    conn.connect();
    // Code for downloading file from the url
  }
  lock.release();
}
```

Fig. 3: Example of downloading a file in Android

that exercises location update frequency of the app. Such tests, however, are not easy to write. For instance, testing the impact of location update by distance requires tests that mock the location. To our knowledge, none of the state-of-the-art Android testing tools are able to generate tests with mocked object. Thereby, developers should manually write such test cases.

Failing to unregister the location listener and listening for a long time consumes a lot of battery power and might lead to location data underutilization [42]. For example in Figure 2, `listener` keeps listening for updates, even if `TrackActivity` is paused in the background. Such location updates are redundant, as the activity is not visible. RLU mutants delete the listener deactivation by commenting the invocation of `removeUpdates` method. This class of mutants can be performed in `onPause` method (RLU_P), `onDestroy` method (RLU_D), or anywhere else in the code (RLU). Killing RLU mutants, specially RLU_D and RLU_P, requires test cases that instigate transitions between activity lifecycle and service lifecycle to ensure that registering/unregistering of location listeners are performed properly under different use cases.

C. Connectivity Mutation Operators

Connectivity-related mutation operators can be divided to network-related, which engage the WiFi or Radio, and bluetooth-related. Mutation operators in both sub-categories mimic energy anti-patterns that unnecessarily utilize WiFi, Radio, and Bluetooth hardware components, which can have a significant impact on the battery discharge rate.

1) Network Mutation Operators: Searching for a network signal is one of the most power-draining operations on mobile devices [11]. As a result, an app needs to first check for connectivity before performing any network operation to save battery, i.e., not forcing the mobile Radio or WiFi to search for a signal, if there is none available. For instance, the code snippet of Figure 3 shows an Android program that checks for connectivity first, and then connects to a server at a particular URL and downloads a file. This can be performed by calling the method `isConnected` of `NetworkInfo`. FCC operator mutates the code by replacing the return value of `isConnected` with true (FCC_R), or adds a conditional statement to check connectivity before performing a network task, if it is not already implemented by the app (FCC_A).

FCC_R

```
if(true){
  HttpURLConnection conn = url.openConnection();
  conn.connect();
  // Code for downloading file from the url
}
```

FCC operators are hard to kill, as they require tests that exercise an app both when it is connected to and disconnected from a network. To that end, tests need to either mock the network connection or programmatically enable/disable network connections.

Another aspect of network connections related to energy is that energy cost of communication over cellular network is substantially higher than WiFi. Therefore, developers should adjust the behavior of their apps depending on the type of network connection. For example, downloads of significant size should be suspended until there is a WiFi connection.

UCW operator forces the app to perform network operations only if the device is connected to cellular network (UCW_C) or WiFi (UCW_W) by adding a conditional statement. For UCW_W, network task is performed only when there is a WiFi connection available. For UCW_C, on the other hand, the mutation operator disables the WiFi connection and checks if a cellular network connection is available to perform the network task. Therefore, killing both mutants requires testing an app using both types of connections.

UCW_W

```
WifiManager manager = getSystemService("WIFI_SERVICE");
if(manager.isWifiEnabled()){
  HttpURLConnection conn = url.openConnection();
  conn.connect();
  // Code for downloading file from the url
}
```

2) Bluetooth Mutation Operators: Figure 4 illustrates a code snippet that searches for paired Bluetooth devices in Android. Device discovery is a periodic task and since it is a heavyweight procedure [4], frequent execution of discovery

```
1  public void discover(int scan_interval){
2    BluetoothAdapter blue =  BluetoothAdapter.
         getDefaultAdapter();
3    private Runnable discovery = new Runnable() {
4      public void run() {
5        blue.startDiscovery();
6        handler.postDelayed(this, scan_interval);
7      }
8    };
9    handler.postDelayed(discovery, 0);
10   connectToPairedDevice();
11   TransferData();
12   handler.removeCallbacks(blue);
13 }
```

Fig. 4: Example of searching for Bluetooth devices in Android

process for Bluetooth pairs can consume high amounts of energy. Therefore, developers should test the impact of discovery process on the battery life.

FBD mutation operator increases the frequency of discovery process by changing the period of triggering the callback that performs Bluetooth discovery to 0, e.g., replacing `scan_interval` with 0 in line 6 of Figure 4. Apps can repeatedly search for Bluetooth pairs using `Handlers` (realized in FBD_H), as shown in Figure 4, or `ScheduledThreadPoolExecuter` (realized in FBD_S), an example of which is available at [10]. Killing FBD mutants requires test cases not only covering the mutated code, but also running *long enough* to show the impact of frequency on power consumption.

Failing to stop the discovery process when the Bluetooth connections are no longer required by the app keeps the Bluetooth awake and consumes energy. RBD operator deletes the method call `removeCallbacks` for a task that is responsible to discover Bluetooth devices, causing redundant Bluetooth discovery. Killing RBD mutants may require tests that transit between Android's activity or service lifecycle states, e.g., trigger termination of an activity/service without stopping the Bluetooth discovery task.

D. Wakelock Mutation Operators

Wakelocks are mechanisms in Adnroid to indicate that an app needs to keep the device (or part of the device such as CPU or WiFi) awake. Inappropriate usage of wakelocks can cause no-sleep bugs [49] and seriously impact battery life and consequently user experience [43]. Developers should test their apps under different use-case scenarios to ensure that their strategy of acquiring/releasing wakelocks does not unnecessarily keep the device awake.

Wakelock-related mutation operators delete the statements responsible to release the acquired wakelock. Depending on the component that acquires a wakelock (e.g., CPU or WiFi), the type of defining wakelock (e.g., `PowerManager`, `WakefulBroadcastReceiver`), and the point of releasing wakelock. We identified and developed support for 8 wakelock-related mutation operators (details and examples can be found at [10]).

E. Display Mutation Operators

Some apps, such as games and video players, need to keep the screen on during execution. There are two ways of keeping the screen awake during execution of an app, namely using screen flags (e.g., `FLAG_KEEP_SCREEN_ON`) to force the screen to stay on, or increasing the timeout of the screen.[2]

Screen flags should only be used in the activities, not in services and other types of components [6]. In addition, if an app modifies the screen timeout setting, these modifications should be restored after the app exits. As an example of display-related mutation operators, MST adds statements to activity classes to increase the screen timeout to the maximum possible value. For MST, there is also a need to modify the manifest file in order to add the permission to modify settings.

MST changes to source code

```
Settings.System.putInt(getContentResolver(),
    "SCREEN_OFF_TIMEOUT", Integer.MAX_VALUE);
```

MST changes to manifest file

```
<uses-permission android:name="android.permission.
     WRITE_SETTINGS"/>
```

F. Recurring Callback and Loop Mutation Operators

Recurring callbacks, e.g., `Timer`, `AlarmManager`, `Handler`, and `ScheduledThreadPoolExecuter`, are frequently used in Android apps to implement repeating tasks. Poorly designed strategy to perform a repeating task may have serious implications on the energy usage of an app [8], [12]. Similarly, loop bugs occur when energy greedy APIs are repeatedly, but unnecessarily, executed in a loop [20], [49].

One of the best practices of scheduling repeating tasks is to adjust the frequency of invocation depending on the battery status. For example, if the battery level drops below 10%, an app should decrease the frequency of repeating tasks to conserve the battery for a longer time. While HFC class of mutation operators unconditionally increases the frequency of recurring callbacks, BFA operators do this only when the battery is discharging. Therefore, the BFA mutants can be killed only when tests are run on a device with low battery or the battery status is mocked. Depending on the APIs that are used in an app for scheduling periodic tasks, we implemented 8 mutation operators of type BFA. As with some of the other operators, details and examples can be found at [10].

G. Sensor Mutation Operators

Sensor events, such as those produced by *accelerometer* and *gyroscope*, can be queued in the hardware before delivery. Setting delivery trigger of sensor listener to low values interrupts the main processor at highest frequency possible and prevents it to switch to lower power state. This is particularly so, if the sensor is a *wake-up* sensor [14]. The events generated by

[2]Since screen wakelocks are deprecated as of Android version 3.2, we only considered screen flags and screen timeout in the design of our operators.

```
1  private SensorEventListener listener;
2  private SensorManager manager;
3  protected void onCreate(){
4    listener = new SensorEventListener();
5    manager = getSystemService("SENSOR_SERVICE");
6    Sensor acm = manager.getDefaultSensor(
7        "TYPE_ACCELEROMETER");
8    manager.registerListener(listener, acm,
9        "SENSOR_DELAY_NORMAL", 5*60*1e6);
10 }
11 protected void onPause(){
12   super.onPause();
13   manager.unregisterListener(listener);
14 }
```

Fig. 5: Example of utilizing sensors in Android

wake-up sensors cause the main processor to wake up and can prevent the device from becoming idle.

In the apps that make use of sensors, tests are needed to ensure that the usage of sensors is implemented in an efficient way. FDSL operator replaces the trigger delay—last parameter in method `registerListener` in line 7 of Figure 5—to *0*, and changes the wake-up property of the sensor in line 6.

FDSL

```
Sensor acm = manager.getDefaultSensor("TYPE_ACCELEROMETER"
    , true);
manager.registerListener(listener, acm,
    "SENSOR_DELAY_NORMAL", 0);
```

In addition, apps should unregister the sensors properly, as the system will not disable sensors automatically when the screen turns off. A thread continues to listen and update the sensor information in the background, which can drain the battery in just a few hours [14]. SLUD operator deletes the statements responsible for unregistering sensor listeners in an app. For a test to kill a SLUD mutant, it needs to trigger a change in the state of app (e.g., terminate or pause the app) without unregistering the sensor listener.

IV. Analyzing Mutants

Mutation testing is known to effectively assess the quality of a test suite in its ability to find real faults [19], [36]. However, it suffers from the cost of executing a large number of mutants against the test suite. This problem is exacerbated by considering the amount of human effort required for analysis of the results, i.e., whether the mutants are killed or not, as well as identifying the equivalent mutants [35]. To streamline usage of mutation testing for energy purposes, we propose a generally applicable, scalable, and fully automatic approach for analyzing the mutants, which relies on a novel algorithm for comparing the power traces obtained from execution of test cases.

A. Killed Mutants

During the execution of a test, power usage can be measured by a power monitoring tool, and represented as a *power trace*—a temporal sequence of power values. A power trace consists of hundreds or more spikes, depending on the sampling rate of the measurement, and can have different shapes, based on the energy consumption behavior.

Algorithm 1: Energy-Aware Mutation Analysis

Input: T Test suite, A Original app, A' Mutant
Output: Determine if a mutant is killed or lived

1 **foreach** $t_i \in T$ **do**
2 $\quad isKilled_i = false$;
3 $\quad P_A = getTrace(A, t_i, 30)$;
4 $\quad \vec{p_{A'}} = getTrace(A', t_i, 1)$;
5 $\quad \vec{r_A} = findRepresentativeTrace(P_A)$;
6 $\quad \alpha = computeThreshold(\vec{r_A}, P_A \setminus \vec{r_A})$;
7 $\quad distance = computeDistance(\vec{r_A}, \vec{p_{A'}})$;
8 $\quad$ **if** $distance > \alpha$ **then**
9 $\quad\quad isKilled_i = true$;

Figure 6 shows the impact of a subset of our mutation operators on the power trace of Sensorium [13]—an app that collects sensor values of a device (e.g., Radio, GPS, and WiFi) and reports them to the user. Figure 6a is the power trace of executing a test on the original version of this app. Figures 6b-d show power traces of the same test after the app is mutated by RLU, FSW_H, and MSB operators, respectively. We can observe that these mutation operators have different impacts on the power trace of the test case.

We have developed a fully-automatic oracle, that based on the differences in the power traces of a test executed on the original and mutant versions of an app, is able to determine whether the mutant was killed or not. Algorithm 1 shows the steps in our approach.

The algorithm first runs each test, t_i, 30 times on the original version of an app, A, and collects a set of power traces, P_A (line 3). The repetition allows us to account for the noise in profiling. Since our analysis to identify a threshold is based on a statistical approach, we repeat the execution 30 times to ensure a reasonable confidence interval[3].

The algorithm then runs each test t_i on the mutant, A', and collect its power trace $\vec{p_{A'}}$ (line 4). Alternatively, for higher accuracy, the test could be executed multiple times on the mutant. However, our experiments showed that due to the substantial overlap between the implementation of the original and mutant versions of the app, repetitive execution of a test on the original version of the app already accounts for majority of the noise in profiling.

Following the collection of these traces, the algorithm needs to determine how different is the power trace of mutant, $\vec{p_{A'}}$, in comparison to the set of power traces collected from the original version of the app, P_A. To that end, the algorithm first has to determine the extent of variation, α, in the 30 energy traces of P_A that could be considered "normal" due to the noise in profiling.

One possible solution to compute this variation is to take their Euclidean distances. However, Euclidean distance is very sensitive to warping in time series [37]. We observed that power traces of a given test on the same version of the app

[3] According to *Central Limit Theorem*, by running the experiments at least thirty times, we are able to report the statistical values within a reasonable confidence interval [50].

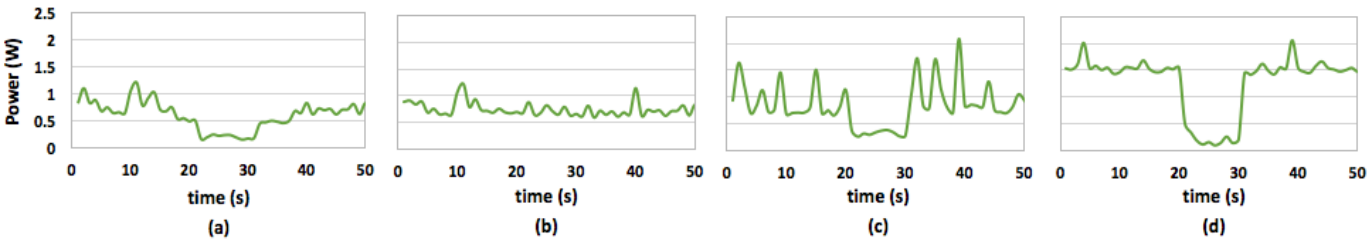

Fig. 6: (a) Baseline power trace for the Sensorium [13] app, and the impact of (b) RLU, (c) FSW_H and (d) MSB mutation operators on the power trace

could be similar in the shape, but locally out of phase. For example, depending on the available bandwidth, quality of the network signal, and response time of the server, downloading a file can take 1 to 4 seconds. Thereby, the power trace of the test after downloading the file might be in the same shape, but shifted and warped in different repetitions of the test case.

To account for inevitable distortion in our power measurement over time, we measure the similarity between power traces by computing the *Dynamic Time Warping (DTW)* distance between them. DTW is an approach to measure the similarity between two time series, independent of their possible non-linear variations in the time dimension [23]. More specifically, DTW distance is the optimal amount of alignment one time series requires to match another time series.

Given two power traces $\vec{P_1}\,[1 \ldots n]$ and $\vec{P_2}\,[1 \ldots m]$, DTW leverages a dynamic programming algorithm to compute the minimum amount of alignments required to transform one power trace into the other. It constructs an $n \times m$ matrix D, where $D[i,j]$ is the distance between $\vec{P_1}\,[1 \ldots i]$ and $\vec{P_2}\,[1 \ldots j]$. The value of $D[i,j]$ is calculated as follows:

$$D[i,j] = |\, P_1[i] - P_2[j] \,| + \min \begin{cases} D[i-1,j] \\ D[i,j-1] \\ D[i-1,j-1] \end{cases} \quad (1)$$

The DTW distance between $\vec{P_1}$ and $\vec{P_2}$ is $D[n,m]$. The lower is the DTW distance between two power traces, the more similar in shape they are.

To determine α, the algorithm first uses DTW to find a representative trace for A, denoted as $\vec{r_A}$ (line 5). It does so by computing the mutual similarity between 30 instances of power trace and choosing the one that has the highest average similarity to the other instances.

Once Algorithm 1 has derived a representative power trace, it lets α to be the upper bound of the 95% confidence interval of the mean distances between the representative power trace and the remaining 29 in P_A (line 6). This means that if we run t_i on A again, the DTW distance between its power trace and representative trace has a 95% likelihood of being less than α different.

Finally, Algorithm 1 computes the DTW distance between $\vec{r_A}$ and $\vec{p_{A'}}$ (line 7). If $distance$ is higher than α, the variation is higher than that typically caused by noise for test t_i, and the mutant is killed; Otherwise, the mutant lives (lines 8- 9).

B. Equivalent and Stillborn Mutants

An *equivalent mutant* is created when a mutation operator does not impact the observable behavior of the program. To determine if a program and one of its mutants are equivalent is an undecidable problem [24]. However, well-designed mutation operators can moderately prevent creation of equivalent mutants. Our mutation operators are designed based on the defect model derived from issue trackers and best practices related to energy. Therefore, they are generally expected to impact the power consumption of the device.

In rare cases, however, mutation operators can change the program without changing its energy behavior. For example, the arguments of a recurring callback that identifies the frequency of invocation may be specified as a parameter, rather than a specific value, e.g., `scan_interval` at line 6 of Figure 4. If this parameter is initialized to 0, replacing it with 0 by *μDroid*'s FBD operator creates an equivalent mutant. As another example, LRP_C can generate equivalent mutants. LRP_C mutants change the provider of location data (e.g., first parameter of `requestLocationUpdates` at line 11 in Figure 2) to `"GPS"`. Although location listeners can be shared among different providers, each listeners can be registered for specific provider once. As a result, If the app already registers a listener for `"GPS"`, LRP_C would create an equivalent mutant.

To avoid generation of equivalent mutants, *μDroid* employs several heuristics and performs an analysis on the source code to identify the equivalent mutants. For example, *μDroid* performs an analysis to resolve the parameter's value and compares it with the value that mutation operator wants to replace. If the parameter is initialized in the program and its value is different from the replacement value, *μDroid* generates the mutant. Otherwise, it identifies the mutant as equivalent and does not generate it.

The Eclipse plugin realizing *μDroid* is able to recognize *stillborn mutants*—those that make the program syntactically incorrect and do not compile. *μDroid* does so by using Eclipse JDT APIs to find syntax errors in the working copy of source code, and upon detecting such errors, it rolls back the changes.

V. Evaluation

In this section, we present experimental evaluation of *μDroid* for energy-aware mutation testing. Specifically, we investigate the following three research questions:

TABLE II: Mutation analysis of each class of mutation operators for subject apps.

Operator ID		LUF	LRP	RLU	LKL	WRDW	HPW	FCC	FSW	RWS	UCW	LTC	DRD	MST	MSB	UAB	FDB	RBD	HFC	RRC	RAF	BFA	ILI	SLUD	FDSL
Total Mutants		9	14	7	1	1	1	3	1	1	8	3	1	5	5	1	1	1	8	8	1	8	20	4	4
T_i	Killed Mutants	5	7	0	1	0	0	0	1	0	4	3	1	0	5	0	1	0	8	0	0	0	20	2	4
	Alive Mutants	4	7	7	0	1	1	3	0	1	4	0	0	5	0	1	0	1	0	8	1	8	0	2	0
T_e	Killed Mutants	6	9	6	1	1	0	2	1	1	8	3	1	5	5	1	1	1	8	5	0	8	20	4	4
	Alive Mutants	3	5	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	3	1	0	0	0	0

TABLE III: Subject apps and test suites.

Apps	LoC	Number of Mutants	Number of Tests		Coverage		Mutation Score	
			T_i	T_e	T_i	T_e	T_i	T_e
aMetro	26,878	17	28	32	59%	59%	35%	76%
OpenCamera	15,064	21	34	36	58%	58%	80%	95%
Jamendo	8,709	21	25	30	65%	65%	57%	95%
a2dp.Vol	6,670	16	29	33	66%	66%	42%	78%
Sensorium	3,288	41	25	31	69%	69%	55%	89%

RQ1. *Quality and Effectiveness*: How effective is *μDroid* in evaluating the quality of test suites for energy testing? Does *μDroid* help developers with creating better tests for revealing energy defects?

RQ2. *Accuracy:* How accurate is *μDroid*'s oracle in determining whether tests kill the energy mutants or not?

RQ3. *Performance*: How long does it take for *μDroid* to create and analyze the mutants?

A. Experimental Setup and Implementation

Subject Apps: To evaluate *μDroid* in practice, we randomly downloaded 20 apps from various categories of *F-Droid* open source repository, applied *μDroid* on them, and selected 5 apps for which *μDroid* was able to generate at least 15 mutants. Note that the number of mutants generated by *μDroid* depends on the Android APIs and libraries used by the app. *μDroid* injected a total of 116 energy-aware mutation operators in the subject apps, distributed among them as shown in Table III.

Power and Coverage Measurement: To profile power consumption of the device during execution of test cases, we used *Trepn* [22]. Trepn is a profiling tool developed by *Qualcomm* that collects the exact power consumption data from sensors embedded in the chipset. Trepn is reported to be highly accurate, with an average of 2.1% error in measurement [16]. To collect power traces, we configured Trepn to read power consumption data at a 100 millisecond interval. The mobile device used in our experiments was Google Nexus 6, running Android version 6.0.1.

Power measurement is sensitive to the workload of the test as well as environmental factors. We used reproducible tests in *Robotium* [17] format to run identical tests on both original and mutant versions of the app. Table III shows the statement coverage of test suites for subject apps, collected using *EMMA* [5]. Although the statement coverage of test suites are not 100%, we ensured that all the mutated parts of an app are covered by test suites of subject apps.

Plugin Implementation: Our Eclipse plugin is publicly available [10] and supports first order and higher order [34] mutation testing. It takes the source code of the original app, parses it to an AST, traverses the AST to find the patterns specified by mutation operators, and creates a mutant for each pattern found in the source code. For an efficient traversal of the AST, the plugin implements mutation operators based on *visitor pattern*. For example, instead of traversing all nodes of the AST to mutate one argument of a specific API call mentioned in the pattern, we only traverse AST nodes of type *MethodInvocation* to find the API call in the code and mutate its argument.

In addition to changes that are applied to the source code, some mutation operators need modification in the XML files of the app. For instance, mutation operator MST adds statements to the source code to change phone settings, which entails adding the proper access permissions to the app's *manifest* file.

B. RQ1: Quality and Effectiveness

We wanted to evaluate (1) if *μDroid* can help engineers determine how effective is their test suite for revealing energy defects, and (2) whether *μDroid* would allow the engineers to improve the quality of test suites. To that end, we first created a test suite T_i for each of the subject apps, such that it covered all the mutated parts of the program. Since the parts of the program mutated by *μDroid* are generally energy greedy, we believe T_i is in fact representative of a test suite that an energy-conscious engineer might develop for the apps. In other words, a skilled developer without access to *μDroid*, but with knowledge of the energy greedy Android APIs, is likely to develop a test suite executing all the energy greedy parts of the program that have been mutated by our approach. Tables II and III show the result of running T_i on the subject

apps. As we can see, while the test suite is able to kill some of the mutants, many of the mutants stay alive, even though they were executed.

The fact that so many of the mutants could not be killed, prompted us to explore the deficiencies in our initial test suites. We found lots of opportunities for improving our test suites, such as development of new test cases that exercised the apps using certain sequences of lifecycle callbacks, or under certain settings (e.g., particular network connectivity). We, thus, developed new tests, which together with T_i, resulted in an enhanced test suite T_e for each app. As shown in Tables II and III, T_e was able to kill a substantially larger number of mutants in all apps.

These results confirm the expected benefits of *μDroid* in practice. While T_e and T_i achieve comparable code coverage and cover the same mutated parts of the program, T_e was substantially more effective at killing the mutants. Thereby, *μDroid* produces *strong* mutants, which can effectively challenge the developers in designing better tests.

It is worth noting that even our enhanced test suite could not kill all the mutants. For instance, we were not able to kill LUF and LRP mutants in *a2dp.Vol* app. By investigating the source code and behavior of this app, we found that accessing the `LocationManager` and requesting for location updates are performed in a short-lived service, thereby the impact of mutation operators on the power trace is negligible and can not be detected by *μDroid*. Similarly, our test suite was not able to kill the RAF mutant and a subset of RRC mutants. These mutants comment out statements that unregister callback methods of an app, such as that shown in line 12 of Figure 4. As the frequency of recurring callbacks was initially low for a subset of subject apps (e.g., a `Handler` callback in *Sensorium* app was invoked every 30 seconds), they required test cases running for several minutes to show the impact on power trace.

C. RQ2: Accuracy

To assess the accuracy of the *μDroid*'s oracle, we first manually built the ground-truth by comparing the shape of power traces for each original app and its mutants. To build the ground truth, we asked two mobile developers, both with substantial Android development experience, neither of whom is an author of this paper, to visually determine if power traces of the original and mutant versions are similar. Visually comparing power traces for similarity in their shape, even if they are out of phase (i.e., shifted, noisy), is an easy, albeit time consuming, task for humans. In case of disagreement, we asked a third developer to compare the power traces.

For each mutant, we only considered tests that executed a mutated part of the program, but not necessarily killed the mutant, and calculated the following metrics:

- *False Positive*: If the ground-truth identifies a mutant as alive, while oracle considers it as killed;
- *False Negative*: If the ground-truth identifies a mutant as killed, while oracle considers it as alive;
- *True Positive*: If the ground-truth and oracle agree a mutant is killed; and
- *True Negative*: If the ground-truth and oracle agree a mutant is alive.

TABLE IV: Accuracy of *μDroid*'s oracle on the subject apps.

Apps	aMetro	Open Camera	Jamendo	a2dp.Vol	Sensorium
Accuracy	92%	95%	100%	91%	93%
Precision	90%	100%	100%	100%	91%
Recall	100%	94%	100%	88%	100%
F-measure	95%	97%	100%	94%	95%

TABLE V: Eclipse plugin analysis time.

Apps	Avg time per mutant (s)	Total time for all mutants(s)
aMetro	0.95	29.74
OpenCamera	0.4	15.22
Jamendo	0.58	10.76
a2dp.Vol	0.24	7.12
Sensorium	0.33	9.56
Average	0.5	14.48

Table IV shows the accuracy of *μDroid*'s oracle for the execution of all tests in T_e on all the subject apps. The results demonstrate an overall accuracy of 94.6% for all the subject apps. Additionally, we observed an average precision of 97% and recall of 95% for the *μDroid*'s oracle. We believe an oracle with this level of accuracy is acceptable for use in practice.

D. RQ3: Performance

To answer this research question, we evaluated the time required for *μDroid* to generate a mutant as well as the time required to determine if the mutant can be killed. We ran the experiments on a computer with 2.2 GHz Intel Core i7 processor and 16 GB DDR3 RAM.

To evaluate the performance of the Eclipse plugin that creates the mutants, we measured the required time for analyzing the code, finding operators that match, and applying the changes to code. From Table V, we can see that *μDroid* takes less than 1 second on average to create a mutant, and 14.4 seconds on average to create all the mutants for a subject app.

To evaluate the performance of oracle, we measured the time taken to determine if tests have killed the mutants. Figure 7 shows the box plot of the time taken to analyze the power trace of all tests from T_e executed on all mutant versions of the 5 subject apps. Form these results we can see that the oracle runs fast; it is able to make a determination as to whether a test is able to kill all of the mutants for one of our subject apps in less than a few seconds.

The analysis time for each test depends on the size of power trace, which depends on the number of power measurements sampled during the test's execution. To confirm the correlation between analysis time and the size of power trace, we computed their Pearson Correlation Coefficient, a.k.a., *Pearson's r* correlation. It measures the linear relationship between two

variables and takes a value between -1 and +1 inclusive, where a value of +1 indicates strong positive correlation and a value -1 indicates strong negative correlation. From the Pearson's r values at the bottom of Figure 7, we can see there is a strong correlation between analysis time and the size of power trace.

VI. Related Work

Our research is related to prior work on mutation testing as well as approaches aimed at identifying energy inefficiencies in mobile apps.

Mutation Testing: Mutation testing has been widely used in testing programs written in different languages, such as Fortran [38], C [25], C# [28], Java [44], and Javascript [48], as well as testing program models and specifications [29], [47]. However, there is a dearth of research on mutation testing for mobile applications, specifically Android apps.

Mutation operators for testing Android apps were first introduced by Deng and colleagues [26], [27], where they proposed eleven mutation operators specific to Android apps. They designed mutation operators based on the app elements, e.g., Intents, activities, widgets, and event handlers. Unlike their operators that are designed for testing functional correctness, our operators are intended for energy testing. Therefore, our mutation operators are different from those proposed in [27]. In addition, they followed a manual approach to analyze the generated mutants, rather than our automatic technique for mutation analysis.

In our previous work [33], we presented an energy-aware test-suite minimization approach and used mutation testing to evaluate the effectiveness of the reduced tests in revealing energy issues. Four out of twenty eight mutation classes presented here, namely WRDC, WRDW, ILI, and HFC, were used to evaluate our prior research. Our prior research, however, did not apply those mutation operators automatically, nor was it able to automatically determine whether the mutants were killed through comparison of power traces.

Green Software Engineering: In recent years, automated approaches for analysis [30], [31], [42], [52], testing [20], refactoring [21], [46], and repair [40], [41] of programs have been proposed by researchers to help developers produce more energy efficient apps.

Liu et al. [42] identified missing sensors and wakelock deactivation as two root causes of energy inefficiencies in Android apps. They proposed a tool, called *GreenDroid*, that can automatically locate these two problems in apps. They have also empirically studied the patterns of wakelock utilization and the impact of wakelock misuse in real-world Android apps [43]. Banerjee and Roychoudhury [21] proposed a set of energy efficiency guidelines to re-factor Android apps for better energy consumption. These guidelines include fixing issues such as sub-optimal binding and nested usage of resources, as well as resource leakage. A subset of our operators are inspired by the energy anti-patterns that are described in these works.

Gupta and colleagues [31] proposed a framework to identify common patterns of energy inefficiencies in power traces, by clustering power traces of a Windows phone running different programs over a period of time. Unlike their approach, we compare power traces of executing a test on two different versions of an app, knowing one is mutated, to determine whether they are different.

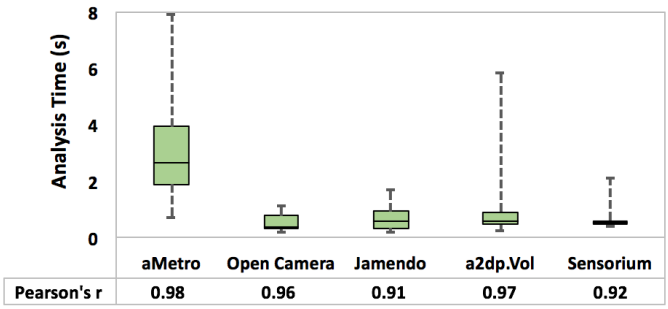

Fig. 7: μ*Droid*'s oracle analysis time

To the best of our knowledge, this paper is the first to propose an energy-aware mutation testing for Android. Our research is orthogonal to other energy-aware testing approaches, such as test generation [20] and regression testing [33], [39], as it helps them to evaluate the quality of the test suites.

VII. Concluding Remarks

Energy efficiency is an important quality attribute for mobile apps. Naturally, prior to releasing apps, developers need to test them for energy defects. Yet, there is a lack of practical tools and techniques for energy testing.

In this paper, we presented μ*Droid*, a framework for energy-aware mutation testing of Android apps. The novel suite of mutation operators implemented in μ*Droid* is designed based on an energy defect model we constructed through an extensive study of various sources, such as open source issue trackers and Android API documentation. μ*Droid* provides an automatic oracle for mutation analysis that compares power traces collected from execution of tests to determine if mutants are killed. Our experiences with μ*Droid* on real-world Android apps corroborate its ability to help the developers evaluate the quality of their test suites for energy testing. μ*Droid* challenges the developers to design tests that are more likely to reveal energy defects.

Currently, we are considering several directions for future work. The dual problem of what we have studied in this paper, seeding energy defects into apps, is automatic repair of apps with energy defects. We are planning to implement an approach to automatically fix Android programs for energy inefficiencies and investigate the extent to which such repairs improve energy consumption. Additionally, given that none of the existing automated Android testing tools are able to generate tests that are sufficiently sophisticated to kill many of the mutants produced by μ*Droid*, we believe the next logical step is the development of test generation techniques suitable for energy testings.

REFERENCES

[1] "Android api reference," https://developer.android.com/reference/packages.html.

[2] "Android developers guide," https://developer.android.com/training/index.html.

[3] "Android open source project issue tracker," https://code.google.com/p/android/issues.

[4] "Bluetoothadapter," https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html.

[5] "Emma java code coverage tool," http://emma.sourceforge.net/.

[6] "Keeping the device awake," https://developer.android.com/training/scheduling/wakelock.html.

[7] "Location manager strategies," https://developer.android.com/guide/topics/location/strategies.html.

[8] "Monitoring the battery level and charging state," https://developer.android.com/training/monitoring-device-state/battery-monitoring.html.

[9] "omim app," https://github.com/mapsme/omim.

[10] "Project website," http://www.ics.uci.edu/~seal/mudroids.

[11] "Reducing network battery drain," https://developer.android.com/topic/performance/power/network/index.html.

[12] "Scheduling repeating alarms," https://developer.android.com/training/scheduling/alarms.html.

[13] "Sensorium android app," https://f-droid.org/repository/browse/?fdid=at.univie.sensorium.

[14] "Sensormanager," https://developer.android.com/reference/android/hardware/SensorManager.html.

[15] "sipdroid app," https://github.com/i-p-tel/sipdroid.

[16] "Trepn accuracy report," https://drive.google.com/file/d/0B0V5MRe1lkP3ZGZwaUlhNVFoZUU/view.

[17] "Trepn accuracy report," http://code.google.com/p/robotium/.

[18] "xdadeveloper android general forum," http://forum.xda-developers.com/android/general.

[19] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?[software testing]," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 2005, pp. 402–411.

[20] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 588–598.

[21] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," 2016.

[22] L. Ben-Zur, "Using Trepn Profiler for Power-Efficient Apps," https://developer.qualcomm.com/blog/developer-tool-spotlight-using-trepn-\profiler-power-efficient-apps.

[23] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series." in *KDD workshop*, vol. 10, no. 16. Seattle, WA, 1994, pp. 359–370.

[24] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[25] M. E. Delamaro, J. Maidonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[26] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 2015, pp. 1–10.

[27] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, 2016.

[28] A. Derezinska and A. Szustek, "Tool-supported advanced mutation approach for verification of c# programs," in *Dependability of Computer Systems, 2008. DepCos-RELCOMEX'08. Third International Conference on*. IEEE, 2008, pp. 261–268.

[29] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, "Mutation analysis testing for finite state machines," in *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*. IEEE, 1994, pp. 220–229.

[30] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 389–398.

[31] A. Gupta, T. Zimmermann, C. Bird, N. Nagappan, T. Bhat, and S. Emran, "Mining energy traces to aid in software development: An empirical case study," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 40.

[32] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, no. 4, pp. 279–290, 1977.

[33] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 425–436.

[34] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[35] ——, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[36] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 654–665.

[37] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and information systems*, vol. 7, no. 3, pp. 358–386, 2005.

[38] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[39] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. Halfond, "Integrated energy-directed test suite optimization," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 339–350.

[40] 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*. ACM, 2016, pp. 249–260.

[41] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of guis in android apps: a multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 143–154.

[42] Y. Liu, C. Xu, S.-C. Cheung, and J. Lü, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, 2014.

[43] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications."

[44] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[45] I. Manotas *et al.*, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 237–248.

[46] I. Manotas, L. Pollock, and J. Clause, "Seeds: a software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 503–514.

[47] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 667–676.

[48] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 74–83.

[49] 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*. ACM, 2011, p. 5.

[50] J. Rice, *Mathematical statistics and data analysis*. Nelson Education, 2006.

[51] C. Wilke, S. Richly, S. Gotz, C. Piechnick, and U. Aßmann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *The Internation Conf. on Green Computing and Communications*, 2013.

[52] H. Wu, S. Yang, and A. Rountev, "Static detection of energy defect patterns in android applications," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 185–195.