µDroid: An Energy-Aware Mutation Testing Framework for Android

Reyhaneh Jabbarvand and Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
{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%, thereby making it possible to apply µDroid automatically.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Software Testing, Mutation Testing, Energy Testing, Android

ACM Reference format:
Reyhaneh Jabbarvand and Sam Malek. 2017. µDroid: An Energy-Aware Mutation Testing Framework for Android. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106244

1 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 [73] and developers [65]. 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 [65]. 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* [50, 55]. 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. [57] 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. 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 [63, 75], 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.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106244

We have extensively evaluated µDroid using open-source Android apps. Our experiments show that µDroid is capable of effectively and efficiently evaluating the adequacy of test suites for revealing energy defects. We found statistically significant correlation between mutation scores produced by µDroid and test suites' ability in revealing energy defects. Furthermore, µDroid's automated oracle showed an average accuracy of 94%, making it possible to apply the mutation testing techniques described in this paper in a fully automated fashion. Finally, using µDroid, we identified 15 previously unknown energy defects in the subject apps. Reporting these defects to developers, 11 of them were verified as bugs and 7 are fixed to date, using the patches we provided to developers. This paper makes the following contributions:

- A 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 in an Eclipse plugin, which is publicly available [21].
- A novel automatic oracle for mutation analysis to identify if an energy mutant can be killed by a test suite.
- Experimental results demonstrating the utility of mutation testing for evaluating the quality of test suites in revealing energy defects.

The remainder of this paper is organized as follows. Section 2 provides an overview of our framework. Sections 3 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 4 introduces our automated approach for energy-aware mutation analysis. Section 5 presents the implementation and evaluation of the research. Finally, the paper outlines related research and concludes with a discussion of our future work.

2 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 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. 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

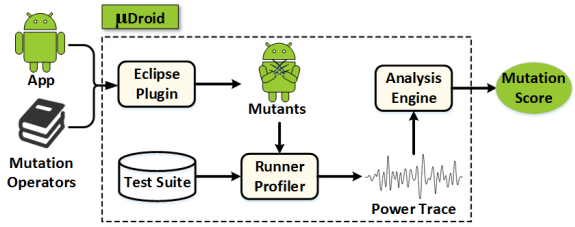

Figure 1: Energy-aware mutation testing framework

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.

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.

3 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.

3.1 Defect Model and Derivation of Operators

Our methodology to collect the energy anti-patterns was a *keyword-based* search approach. We started by crawling the Android Open Source Project issue tracker [3] and XDA Developers forum [36] and searched for posts that have at least one of the following keywords: *energy*, *power*, *battery*, *drain*, and *consumption*. We then manually inspected these posts to find energy-related issues as reported by users and developers. The outcome of this step was a list of energy-related issues and *295* apps that possibly had instances of those issues. We excluded commercial apps from the list, since our study 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 [38, 48, 63, 75] 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 [13] issue 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 [30] 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`.

Table 1: List of proposed energy-aware mutation operators.

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

Investigating the source code of energy-inefficient apps provided us with common mistakes that developers make or mistakes that have severe impact on the energy consumption of apps. In addition, we crawled Android Developers Guide [2] and Android API Reference [1] for best practices related to energy consumption using the aforementioned keywords. This way we found additional energy issues that either happen in specific use-cases that are uncommon among apps, or their impact cannot be readily observed by end users. In total, 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 1 lists our energy-aware mutation operators. We designed and implemented 50 mutation operators (column 3 in Table 1)—corresponding to the identified energy defect patterns, grouped into 28 classes (column 2 in Table 1). 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 [21].

3.2 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 [11]. 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`

```
1  public class TrackActivity extends Activity {
2    private LocationManager manager;
3    private LocationListener listener;
4    protected void onCreate(){
5      manager = getSystemService("LOCATION_SERVICE");
6      listener = new LocationListener(){
7        public void onLocationChanged(){
8          // Use location information to update activity
9        }
10     };
11     manager.requestLocationUpdates("NETWORK", 2*60*1000, 20,
           listener);
12   }
13   protected void onPause(){super.onPause();}
14   protected void onDestroy(){
15     super.onDestroy();
16     manager.removeUpdates(listener);
17   }
18 }
```

Figure 2: Example of obtaining user location in Android

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 4), it shows the presence of at least one test in the test suite 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

```
1  protected void downloadFiles(String link){
2    WifiLock lock = getSystemService().createWifiLock();
3    lock.acquire();
4    URL url = new URL(link);
5    ConnectivityManager manager = getSystemService(
6        "CONNECTIVITY_SERVICE");
7    NetworkInfo nets = manager.getActiveNetworkInfo();
8    if(nets.isConnected()){
9      HttpURLConnection conn = url.openConnection();
10     conn.connect();
11     // Code for downloading file from the url
12   }
13   lock.release();
14 }
```

Figure 3: Example of downloading a file in Android

```
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 }
```

Figure 4: Example of searching for Bluetooth devices in Android

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 [63]. 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.

3.3 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.

3.3.1 Network Mutation Operators. Searching for a network signal is one of the most power-draining operations on mobile devices [22]. 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 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.

FCC_R

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

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
}
```

3.3.2 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, frequent execution of discovery 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 [21]. 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.

3.4 Wakelock Mutation Operators

Wakelocks are mechanisms in Android 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 [71] and seriously impact battery life and consequently user experience. 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 [21]).

3.5 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.

Screen flags should only be used in the activities, not in services and other types of components [10]. 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 and manifest file

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

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

3.6 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 [12, 23]. Similarly, loop bugs occur when energy greedy APIs are repeatedly, but unnecessarily, executed in a loop [38, 71].

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 [21].

3.7 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 [29]. The events generated by 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.

```
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("ACCELEROMETER");
7    manager.registerListener(listener, acm, "DELAY_NORMAL",
           5*60*1e6);
8  }
9  protected void onPause(){
10   super.onPause();
11   manager.unregisterListener(listener);
12 }
```

Figure 5: Example of utilizing sensors in Android

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.

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[29]. 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.

FDSL

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

4 ANALYZING MUTANTS

Mutation testing is known to effectively assess the quality of a test suite in its ability to find real faults [37, 57]. 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 [55]. 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.

4.1 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.

Figure 6 shows the impact of a subset of our mutation operators on the power trace of Sensorium [24]—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).

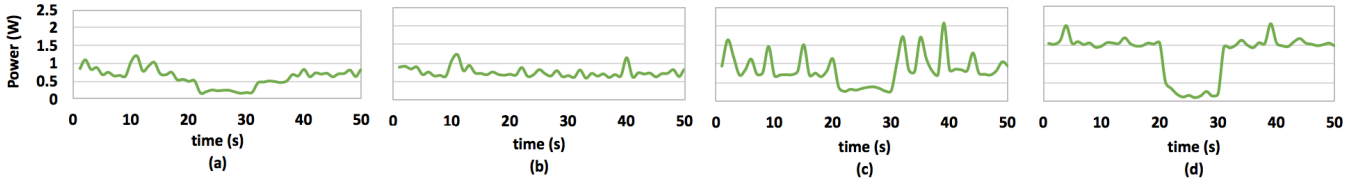

Figure 6: (a) Baseline power trace for Sensorium [24], and the impact of (b) RLU, (c) FSW_H and (d) MSB mutation operators on the power trace

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[1].

The algorithm then runs each test t_i on the mutant, A', and collects 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 [58]. We observed that power traces of a given test on the same version of the app 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 [41]. 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.

[1] 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 [72].

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 ∈ T do
2     isKilled_i = false;
3     P_A = getTrace(A, t_i, 30);
4     p_A' = getTrace(A', t_i, 1);
5     r_A = findRepresentativeTrace(P_A);
6     α = computeThreshold(r_A, P_A \ r_A);
7     distance = computeDistance(r_A, p_A');
8     if distance > α then
9         isKilled_i = true;
```

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).

4.2 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 [42]. 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 listener 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 *still-born 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.

5 EVALUATION

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

- **RQ1.** *Prevalence, Quality, and Contribution*: How prevalent are energy-aware mutation operators in real-world Android apps? What is the quality of energy-aware mutation operators? What is the contribution of each mutant type to the overall mutants generated by μDroid?
- **RQ2.** *Effectiveness*: Does μDroid help developers with creating better tests for revealing energy defects?
- **RQ3.** *Association to Real Faults*: Are mutation scores correlated with test suites' ability in revealing energy faults?
- **RQ4.** *Accuracy*: How accurate is μDroid's oracle in determining whether tests kill the energy mutants or not?
- **RQ5.** *Performance*: How long does it take for μDroid to create and analyze the mutants?

5.1 Experimental Setup and Implementation

Subject Apps: To evaluate μDroid in practice, we randomly collected 100 apps from seventeen categories of *F-Droid* open-source repository. We then selected a subset of the subject apps that satisfied the following criteria: (1) We selected apps for which μDroid was able to generate at least 25 mutants and the generated mutants belonged to at least 3 different categories identified in Table 1. (2) We further reduced the apps to a subset for which we were able to find at least one commit related to fixing an energy defect in their commit history. (3) Finally, to prevent biasing our results, we removed apps that were among the 59 apps we studied to derive the energy defect model, and eventually our operators. At the end, we ended up with a total of 9 apps suitable for our experiments. μDroid injected a total of 413 energy-aware mutation operators in these apps, distributed among them as shown in Table 2.

Mutant Generation: We used μDroid to generate energy mutants. Our Eclipse plugin is publicly available [21] and supports both first-order and higher-order mutation testing [54]. 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 the *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 require modification in the XML files of the app. For instance, mutation operators MST and UCW_C add statements to the source code to change phone or WiFi settings, requiring the proper access permissions to be added to the app's *manifest* file.

Additionally, we compare energy mutants generated by μDroid with mutants generated by Major [56] and the Android mutation framework developed by Deng et al. [45].[2]

Power Measurement: The mobile device used in our experiments was Google Nexus 6, running Android version 6.0.1. To profile power consumption of the device during execution of test cases, we used *Trepn* [40]. 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 [32].

Test Suites: We used two set of reproducible tests to evaluate μDroid. The first set includes tests in *Robotium* [33] and *Espresso* [4] format written by mobile app developers, and the second set includes random tests generated by Android Monkey [35]. Both set of tests are reproducible to ensure we are running identical tests on both original and mutant versions of the app.

Faults: To evaluate the association between mutation score and fault detection ability of test suites, we searched the issue tracker and commit history of the subject apps to find the commits related to fixing energy-related faults. As shown in Table 2, we were able to isolate and reproduce 18 energy-related faults for the subject apps.

5.2 RQ1: Prevalence, Quality, and Contribution

To understand the prevalence of energy-aware mutation operators, we first applied μDroid on the 100 subject apps described in Section 5.1. We found that μDroid is able to produce energy mutants for all programs, no matter how small, ranging from 5 to 110, with an average of 28 mutants. This shows that all apps can potentially benefit from such a testing tool.

Table 2 provides a more detailed presentation of results for 9 of the subject apps, selected according to the criteria described in Section 5.1. Here, we also compare the prevalence of energy-aware operators with prior mutation testing tools, namely Major [56], and Android mutation testing tool of Deng et al. [45]. The result of this comparison is shown in Table 2. Overall, μDroid generates much fewer mutants compared to other tools, which is important given the cost of energy mutation testing, e.g., the need to run and collect energy measurements on resource-constrained devices. In total, μDroid generates 413 mutants for the subject apps, thereby producing 99% and 92% less mutants than Major and Deng et al., respectively. *Spearman's Rank Correlation* between the prevalence of energy-aware mutants and mutants produced by other tools suggests that there is no significant monotonic relationship between them: Major ($\rho = -0.28$) and Deng et al. ($\rho = 0.2$) with significance level $p < 0.01$. This is mainly due to the fact that μDroid targets specific APIs, Android-specific constructs, and other resources, such as layout XML files, that are not considered in the design of mutation operators in other tools.

Furthermore, we calculated the number of μDroid mutants that are duplicate of mutants produced by the other tools. Table 2 presents the percentage of duplicate energy-aware mutants under the *dup* columns. Due to the large number of mutants generated by other tools, we used Trivial Compiler Equivalent (TCE) technique [70] to identify a lower bound for duplicate mutants. TCE is

[2]We were not able to use PIT [31], as PIT does not support Android and its mutants are held only in memory, which prevented us from building the mutant APKs.

Table 2: Test suites and mutants generated for subject apps.

Apps	LoC	Faults	Mutants					Test Suites					
			μDroid			Major [56]	Deng et al. [45]	#Tests		Mutant Coverage		Mutation Score	
			#	[56]-dup	[45]-dup	#	#	T_i	T_e	T_i	T_e	T_i	T_e
DSub	43,032	1	31	10%	0%	19,411	1,539	24	34	63%	83%	19%	71%
Openbmap	31,408	3	46	13%	4%	3,374	460	21	33	79%	97%	32%	93%
aMetro	26,868	1	59	3%	0%	13,419	1,102	26	38	77%	94%	37%	88%
GTalk	17,834	3	42	12%	0%	4,505	336	30	42	73%	95%	28%	95%
Ushahidi	16,470	1	86	2%	5%	4,682	368	26	38	62%	97%	37%	97%
OpenCamera	15,064	1	32	17%	6%	16,717	142	32	44	72%	100%	60%	96%
Jamendo	8,709	1	40	8%	3%	3,599	645	18	29	88%	93%	33%	88%
a2dp.Vol	6,670	4	28	4%	4%	4,682	214	25	34	96%	96%	32%	95%
Sensorium	3,228	3	49	18%	0%	1,589	268	28	35	84%	93%	33%	91%

Table 3: Mutation analysis of each class of mutation operators for subject apps.

Operator ID		LUF	LRP	RLU	LKL	WRDC	WRDW	HPW	FCC	FSW	RWS	UCW	LTC	DRD	MST	LBC	MSB	UAB	FDB	RBD	HFC	RRC	RAF	BFA	ILI	SLUD	FDSL
Total		27	52	28	6	5	3	4	11	1	1	26	11	3	46	44	48	3	1	1	16	13	1	30	20	4	8
Equivalent		0	19	0	0	0	0	0	0	0	0	0	0	0	0	7	0	0	0	0	3	0	0	4	0	0	0
Contribution%		7	12	7	1	1	1	1	3	<1	<1	6	3	1	11	11	12	1	<1	<1	4	3	<1	7	5	1	2
T_i	Killed	12	24	0	3	0	0	0	0	1	0	0	8	0	0	14	34	0	1	0	13	0	0	0	12	0	8
	Alive	15	9	28	3	5	3	4	11	0	1	26	3	3	46	23	14	3	0	1	0	13	1	26	8	4	0
T_e	Killed	26	33	26	6	4	3	1	11	1	1	20	9	3	46	28	44	3	1	1	11	12	0	24	17	4	8
	Alive	1	0	2	0	1	0	0	0	0	0	6	2	0	0	9	4	0	0	0	2	1	1	2	3	0	0

a scalable and effective approach to find equivalent and duplicate mutants by comparing the machine code of compiled mutants. In addition to compiled classes, we also considered any difference in the XML files of the mutants, as μDroid modifies layout and manifest files to create a subset of mutants. On average, only 9% and 2% of mutants produced by μDroid are duplicates of the mutants produced by Major and Deng et al., respectively. These results confirm that μDroid is addressing a real need in this domain, as other tools are not producing the same mutants.

Table 3 also shows the contribution of each class of energy-aware mutation operators for subject apps. Display-related mutation operators have the highest contribution (34%), followed by Location-related (27%), Connectivity-related (16%), and Recurring-related (16%) mutation operators. Wakelock-related (3%) and Sensor-related (3%) oeprators have less contribution. These contributions are associated to the power consumption of hardware components, since display, GPS, WiFi, and radio are reported to consume the highest portion of device battery [5]. Finally, μDroid generates no still-born mutants, and only 8% of all the mutants were identified to be equivalent, as shown in Table 3.

To summarize, the results from RQ1 indicate that (1) potentially all apps can benefit from such a testing tool, as μDroid was able to generate mutants for all 100 subject apps, (2) the small number of mutants produced by μDroid makes it a practical tool for energy-aware mutation testing of Android, (3) the great majority of energy-aware mutation operators are unique and the corresponding mutants cannot be produced by previous mutation testing tools, and (4) all operators incorporated in μDroid are useful, as they were all applied on the subject apps, albeit with different degrees of frequency.

5.3 RQ2: Effectiveness

To evaluate whether μDroid can help developers to improve the quality of test suites, we asked two mobile app developers, both with substantial professional Android development experience at companies such as Google, to create test suites for validating the energy behavior of 9 subject apps. These initial test suites, denoted as T_i, contained instrumented tests to exercise the app under various scenarios. Tables 2 and 3 show the result of running T_i on the subject apps. As we can see, while the initial test suites are able to execute the majority of mutants (high mutant coverage values on Table 2), many of the mutants stay alive (low mutation score on Table 2).

The fact that so many of the mutants could not be killed, prompted us to explore the deficiencies in initial test suites with respect to alive mutants. We found lots of opportunities for improving the initial test suites, such as adding tests with the following characteristics:

- **Exercising sequences of activity lifecycle**: Wakelocks and other resources such as GPS are commonly acquired and released in lifecycle event handlers. Therefore, the only way to test the proper management of resources and kill mutants such as RLU, WRDW, WRDC, and MST, is to exercise particular sequence of lifecycle callbacks. Tests that pause or tear-down activities and then resume or relaunch an app can help with killing such mutant.
- **Manipulate network connection**: A subset of network-related mutation operators, namely FCC, UCW, HPW, and RWS, only change the behavior of the app under peculiar network connectivity. For example, FCC can be killed only when there is no network connectivity, and HPW can be killed by testing the app under a poor WiFi signal condition. Tests that programmatically manipulate network connections are generally effective in killing such mutants.
- **Manipulate Bluetooth or battery status**: None of the UAB, RBD, and BFA mutants were killed by the initial test suites. That is mainly due to the fact that the impact of such mutants is only observable under specific status of Bluetooth and battery. For example, BFAs change the behavior of an app only when the battery is low, requiring tests that can programmatically change or emulate the state of such components.
- **Effectively mock location**: Location-based mutants can be killed by mocking the location. Although changing the location once may cover the mutated part, effectively killing the location mutants, specifically LUF, requires mocking the location several times and under different speeds of movement.
- **Longer tests**: Some mutants, namely LBC, LTC, and RAF, can be killed only if the tests run long enough for their effect to be

Table 4: Accuracy of name's oracle on the subject apps.

Apps	Accuracy	Precision	Recall	F-measure
DSub	95%	95%	97%	96%
Openbmap	91%	94%	91%	92%
aMetro	92%	90%	100%	95%
GTalk	93%	97%	93%	95%
Ushahidi	94%	97%	94%	95%
OpenCamera	95%	100%	94%	97%
Jamendo	100%	100%	100%	100%
a2dp.Vol	91%	100%	98%	94%
Sensorium	93%	91%	100%	95%
Average	94%	96%	96%	95%

observed. For example, if the test tries to download a file and terminates immediately, the impact of LTC forcing an app to wait for a connection being established is not observable on the power trace.

- **Repeating tasks**: A subset of mutants are not killed unless a task is repeated to observe the changes. For example, DRD mutants are only killed if a test tries to download a file multiple times.

We subsequently asked the subject developers to generate new tests with the aforementioned characteristics, which together with T_i, resulted in an enhanced test suite T_e for each app. As shown in Tables 2 and 3, T_e was able to kill substantially more mutants in all apps. These results confirm the expected benefits of µDroid in practice. While T_i achieves a reasonable mutant coverage (80% on average among all subject apps), it was not able to accomplish high mutation score (35% on average). This demonstrates that µDroid produces *strong* mutants (i.e., hard to kill), thereby effectively challenging the developers in designing better tests.

Furthermore, running the enhanced test suites on the 9 subject apps, we were able to find 15 previously unknown energy bugs. After reporting them to the developers, 11 of them have been confirmed as bugs by the developers and 7 of them have been fixed by the submission date of this paper, as corroborated by their issue trackers [6–9, 14–20, 25–28].

5.4 RQ3: Association to Real Faults

If mutation score is a good indicator of a test suite's ability in revealing energy bugs, one would expect to be able to show a statistical correlation between the two. Since calculating such a correlation requires a large number of test suites per fault, we first generated 100 random tests for each app using Android Monkey [35]. For each app, we randomly selected 20 tests from its test suite (consisting of both random and developer-written tests mentioned in the previous section) and repeated the sampling 20 times. That is, in the end, for each subject app we created 20 test suites, each containing 20 tests from a pool of random and developer-written tests.

We then ran each test suite against the 9 subject apps, and marked them as T_{fail}, if the test suite was able to reveal any of its energy faults, or T_{pass}, if the test suite was not able to reveal any of its energy faults. To avoid bias, in this experiment we did not consider the energy faults found by us, rather focused on those that had been found and reported previously. For each T_{fail} and T_{pass}, we computed the mutation score of the corresponding test suite. Finally, for each fault, we constructed test suite pairs of $\langle T_{fail}, T_{pass} \rangle$ and computed the difference in their mutation score, which we refer to as *mutation score difference (MSD)*.

Among a total of 1,257 pairs of $\langle T_{fail}, T_{pass} \rangle$ generated for 18 faults, T_{fail} was able to attain a higher mutation score compared to T_{pass} in 77% of pairs. Furthermore, to determine the strength of correlation between mutation kill score and fault detection, we used one sample t-test, as MSD values were normally distributed and there were no outliers in the dataset (verified with Grubbs' test). Our *null* hypothesis assumed that the average of the MSD values among all pairs equals to 0, while the upper tailed *alternative* hypothesis assumed that it is greater than 0. The result of one sample t-test over 1,257 pairs confirmed that there is a statistically significant difference in the number of mutants killed by T_{fail} compared to T_{pass} (p-value = 9.87E-50 with significance level $p < 0.0001$). Small p-value and large number of samples confirm that the results are unlikely to occur by chance. Note that we removed equivalent and subsumed mutants for MSD calculation to avoid Type I error [69].

5.5 RQ4: Accuracy of Oracle

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 the previously mentioned developers 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 *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 both agree a mutant is killed), and *true negative* (if both agree a mutant is alive) metrics.

Table 4 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% for all the subject apps. Additionally, we observed an average precision of 96% and recall of 96% for the µDroid's oracle. We believe an oracle with this level of accuracy is acceptable for use in practice.

5.6 RQ5: 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 5, we can see that µDroid takes less than 0.5 second on average to create a mutant, and 11.7 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. Table 5 shows the time taken to analyze the power trace of all tests from T_e executed on all mutant versions of the subject apps. From 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, denoted with *PCC* in Table 5. From the PCC values, we can see there is a strong correlation between analysis time and the size of power trace among all subject apps.

6 RELATED WORK

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

Mutation Testing: Mutation testing has been widely used in testing programs written in different languages, such as Fortran [59], C [43], C# [46], Java [64], and Javascript [68], as well as testing program specifications [47, 67] and program memory usage [74]. 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 [44, 45], 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 [45]. In addition, they followed a manual approach to analyze the generated mutants, rather than our automatic technique for mutation analysis.

Green Software Engineering: In recent years, automated approaches for analysis [48, 49, 52, 53, 63, 75], testing [38, 51], refactoring [39, 66], and repair [61, 62] of programs have been proposed by researchers to help developers produce more energy efficient apps.

Liu et al. [63] 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. Banerjee and Roychoudhury [39] 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.

In our previous work [52], we presented an energy-aware test suite minimization approach for Android. To determine the quality of tests, we used a coverage metric that is calculated based on the actual energy cost of executing the tests. This work complements our prior work, as the mutation score produced by µDroid could be used as an alternative metric for minimization of test suites.

Gupta and colleagues [49] provided 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.

To the best of our knowledge, this paper is the first to propose an automated energy-aware mutation testing framework for Android. Our research is orthogonal to other energy-aware testing approaches, such as test generation [38] and regression testing [52, 60], as it helps them to evaluate the quality of the test suites.

7 THREATS TO VALIDITY

External Validity: Random selection of hundred apps to evaluate our research may introduce external threats to validity of results, as they may not be representative of all apps. To mitigate this threat, we selected apps from various categories of F-Droid to investigate RQ1. For investigating the other research questions, we applied the inclusion criteria discussed in Section 5.1. We believe that the resulting 9 apps are sufficient to demonstrate the overall applicability of the approach, yet small enough to afford a detailed description for the reader.

Internal Validity: Power measurement is sensitive to the workload of the test as well as environmental factors. To control for the power measurement threats, we used reproducible tests to run identical tests on both original and mutant versions of an app. We configured Trepn to read power consumption data at a 100 millisecond interval. As power consumption measurements tend to be less accurate when the device is plugged into a power source, we ran the tests on the device over ADB Wireless. Trepn can operate in two modes, app level and system level. Since we used *Intents* to start and stop profiling and Trepn does not provide a mechanism for profiling a specific app via Intents [34], we measured the power consumption of the device during execution of tests. To remove the impact of other apps on the measurements, we disabled and uninstalled all unnecessary apps on the device.

Table 5: Performance analysis of name on the subject apps.

Apps	Time (s)				PCC
	Total	Per Mutant	Analysis	Per Test	
DSub	27.3	1.0	20.74	0.61	0.91
Openbmap	15.9	0.6	28.05	0.85	0.97
aMetro	14.7	0.4	60.08	1.6	0.94
GTalk	18.4	0.8	40.74	0.97	0.92
Ushahidi	8.1	0.2	50.34	1.43	0.95
OpenCamera	5.8	0.3	16.28	0.37	0.96
Jamendo	9.3	0.4	16.24	0.56	0.94
a2dp.Vol	3.5	0.2	19.38	0.57	0.94
Sensorium	3.2	0.1	18.2	0.52	0.9
Average	11.7	0.4	30.5	0.83	-

Construct Validity: To derive anti-patterns, we used a collection of keywords to find apps with energy defect. Although we make no claims that this set of keywords is minimal or complete, prior research has shown that they are frequently used in the issue trackers of apps with energy bugs [63]. We acknowledge that the collection of energy issues identified through our study may not be complete due to this reason. However, we believe the presented energy-aware defect model is the most comprehensive one in the literature to date.

8 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 µDroid, a framework for energy-aware mutation testing of Android apps. The novel suite of mutation operators implemented in µDroid is designed based on an energy defect model, constructed through an extensive study of various sources (e.g., issue trackers, API documentations). µDroid 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 µDroid on real-world Android apps corroborate its ability to help the developers evaluate the quality of their test suites for energy testing. µDroid challenges the developers to design tests that are more likely to reveal energy defects.

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 µDroid, we believe the next logical step is the development of test generation techniques suitable for energy testing.

ACKNOWLEDGMENTS

This work was supported in part by awards CCF-1252644, CNS-1629771, and CCF-1618132 from the National Science Foundation, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

REFERENCES

[1] 2017. Android API Reference. (2017). https://developer.android.com/reference/packages.html
[2] 2017. Android Developers Guide. (2017). https://developer.android.com/training/index.html
[3] 2017. Android Open Source Project issue tracker. (2017). https://code.google.com/p/android/issues
[4] 2017. Android Testing Support Library : Espresso. (2017). https://google.github.io/android-testing-support-library/docs/espresso/
[5] 2017. Battery Life. (2017). https://dl.google.com/io/2009/pres/W_0300_CodingforLife-BatteryLifeThatIs.pdf
[6] 2017. GTalk:issue 279. (2017). https://github.com/Yakoo63/gtalksms/issues/279
[7] 2017. GTalk:issue 280. (2017). https://github.com/Yakoo63/gtalksms/issues/280
[8] 2017. Jamendo:issue 38. (2017). https://github.com/telecapoland/jamendo-android/issues/38
[9] 2017. Jamendo:issue 39. (2017). https://github.com/telecapoland/jamendo-android/issues/39
[10] 2017. Keeping the Device Awake. (2017). https://developer.android.com/training/scheduling/wakelock.html
[11] 2017. Location Manager Strategies. (2017). https://developer.android.com/guide/topics/location/strategies.html
[12] 2017. Monitoring the Battery Level and Charging State. (2017). https://developer.android.com/training/monitoring-device-state/battery-monitoring.html
[13] 2017. omim app. (2017). https://github.com/mapsme/omim
[14] 2017. Openbmap:issue 175. (2017). https://github.com/openbmap/radiocells-scanner-android/issues/175
[15] 2017. Openbmap:issue 176. (2017). https://github.com/openbmap/radiocells-scanner-android/issues/176
[16] 2017. Openbmap:issue 177. (2017). https://github.com/openbmap/radiocells-scanner-android/issues/177
[17] 2017. Openbmapissue :178. (2017). https://github.com/openbmap/radiocells-scanner-android/issues/178
[18] 2017. Openbmap:issue 179. (2017). https://github.com/openbmap/radiocells-scanner-android/issues/179
[19] 2017. Openbmap:issue 184. (2017). https://github.com/openbmap/radiocells-scanner-android/issues/184
[20] 2017. OpenCamera:issue 251. (2017). https://sourceforge.net/p/opencamera/tickets/251/
[21] 2017. Project website. (2017). http://www.ics.uci.edu/~seal/projects/mu_droid/index.html
[22] 2017. Reducing Network Battery Drain. (2017). https://developer.android.com/topic/performance/power/network/index.html
[23] 2017. Scheduling Repeating Alarms. (2017). https://developer.android.com/training/scheduling/alarms.html
[24] 2017. Sensorium Android App. (2017). https://f-droid.org/repository/browse/?fdid=at.univie.sensorium
[25] 2017. Sensorium:19. (2017). https://github.com/fmetzger/android-sensorium/issues/
[26] 2017. Sensorium:20. (2017). https://github.com/fmetzger/android-sensorium/issues/
[27] 2017. Sensorium:issue 17. (2017). https://github.com/fmetzger/android-sensorium/issues/
[28] 2017. Sensorium:issue 18. (2017). https://github.com/fmetzger/android-sensorium/issues/
[29] 2017. SensorManager. (2017). https://developer.android.com/reference/android/hardware/SensorManager.html
[30] 2017. sipdroid app. (2017). https://github.com/i-p-tel/sipdroid
[31] 2017. Trepn Accuracy Report. (2017). http://pitest.org/
[32] 2017. Trepn Accuracy Report. (2017). https://drive.google.com/file/d/0B0V5MRe1lkP3ZGZwaUlhNVFoZUU/view
[33] 2017. Trepn Accuracy Report. (2017). http://code.google.com/p/robotium/
[34] 2017. Trepn Modes. (2017). https://developer.qualcomm.com/forum/qdn-forums/increase-app-performance/trepn-profiler/28172
[35] 2017. UI/Application Excersizer Monkey. (2017). http://developer.android.com/tools/help/monkey.html
[36] 2017. xdadeveloper Android general forum. (2017). http://forum.xda-developers.com/android/general
[37] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?[software testing]. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 402–411.
[38] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 588–598.
[39] Abhijeet Banerjee and Abhik Roychoudhury. 2016. Automated Re-factoring of Android Apps to Enhance Energy-efficiency. (2016).
[40] Liant Ben-Zur. 2017. Using Trepn Profiler for Power-Efficient Apps. https://developer.qualcomm.com/blog/developer-tool-spotlight-using-trepn-profiler-power-efficient-apps. (2017).
[41] Donald J Berndt and James Clifford. 1994. Using Dynamic Time Warping to Find Patterns in Time Series. In *KDD workshop*, Vol. 10. Seattle, WA, 359–370.
[42] Timothy A Budd and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta Informatica* 18, 1 (1982), 31–45.
[43] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. 2001. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering* 27, 3 (2001), 228–247.
[44] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. 2015. Towards mutation analysis of android apps. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 1–10.
[45] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. 2016. Mutation operators for testing Android apps. *Information and Software Technology* (2016).
[46] Anna Derezinska and Anna Szustek. 2008. Tool-supported advanced mutation approach for verification of C# programs. In *Dependability of Computer Systems, 2008. DepCos-RELCOMEX'08. Third International Conference on*. IEEE, 261–268.
[47] SC Pinto Ferraz Fabbri, Márcio Eduardo Delamaro, José Carlos Maldonado, and Paulo Cesar Masiero. 1994. Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*. IEEE, 220–229.
[48] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 389–398.
[49] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, Thirumalesh Bhat, and Syed Emran. 2014. 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, 40.
[50] Richard G. Hamlet. 1977. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering* 4 (1977), 279–290.
[51] Reyhaneh Jabbarvand. 2017. Advancing energy testing of mobile applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 491–492.
[52] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware test-suite minimization for Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 425–436.
[53] Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. 2015. Ecodroid: An approach for energy-based ranking of android apps. In *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. IEEE Press, 8–14.
[54] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393.
[55] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.
[56] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 international symposium on software testing and analysis*. ACM, 433–436.
[57] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. 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, 654–665.
[58] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact indexing of dynamic time warping. *Knowledge and information systems* 7, 3 (2005), 358–386.
[59] Kim N King and A Jefferson Offutt. 1991. A fortran language system for mutation-based software testing. *Software: Practice and Experience* 21, 7 (1991), 685–718.
[60] Ding Li, Yuchen Jin, Cagri Sahin, James Clause, and William GJ Halfond. 2014. Integrated energy-directed test suite optimization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 339–350.
[61] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. 2016. Automated energy optimization of HTTP requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 249–260.
[62] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2015. 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, 143–154.
[63] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Jian Lü. 2014. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering* 40, 9 (2014), 911–940.
[64] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133.
[65] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. 2016. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 237–248.
[66] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 503–514.
[67] Evan Martin and Tao Xie. 2007. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*. ACM, 667–676.
[68] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. Efficient JavaScript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 74–83.
[69] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings*

of the 25th International Symposium on Software Testing and Analysis. ACM, 354–365.
[70] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 936–946.
[71] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. 2011. 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, 5.
[72] John Rice. 2006. *Mathematical statistics and data analysis*. Nelson Education.
[73] Claas Wilke, Sebastian Richly, Sebastian Gotz, Christian Piechnick, and Uwe Aßmann. 2013. Energy Consumption and Efficiency in Mobile Applications: A user Feedback Study. In *The Internation Conf. on Green Computing and Communications*.
[74] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. 2017. Memory mutation testing. *Information and Software Technology* 81 (2017), 97–111.
[75] Haowei Wu, Shengqian Yang, and Atanas Rountev. 2016. Static detection of energy defect patterns in Android applications. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 185–195.