µDROID: An Energy-Aware Mutation Testing Framework for Android

A report as part of the seminar Software testing

Stefan Schott

University Paderborn, Institute for Computer Science,
33098 Paderborn, Germany
sschott@mail.uni-paderborn.de

Abstract. Nowadays mobile apps are becoming more and more popular. With this increase in popularity and energy being a limited resource on mobile devices the demand for developers to produce energy efficient applications rises steadily. Currently there are no specialized energy testing frameworks available, therefore developers purely rely on functional correctness tests. These tests are not necessarily suited to uncover energy defects in an application.
µDROID provides an energy-aware mutation testing framework which is able to evaluate a test suite's ability to uncover energy defects. µDROID provides a set of 50 different energy anti-patterns to test on. Furthermore µDROID can be applied to a test suite completely automatic and assign it a score which represents its ability to uncover energy defects. Besides that µDROID can also help in revealing missing test cases.

Keywords: Android, Energy Testing, Mutation Testing, Software Testing

1 Introduction

Mobile apps are becoming more and more popular these days. Even most desktop programs now provide a mobile version. Since energy is a limited resource on mobile devices the energy efficiency of mobile apps becomes more and more of a concern. Recent surveys [3] show that for 59% of participants (n=782) the most wanted feature for their next smartphone is a longer battery runtime.
Because of a lack of tools for energy testing for Android applications, currently functional correctness test suites are used to uncover energy-related defects. These test suites are not necessarily suited to uncover energy-related defects.
This work is a report about µDROID [8]. µDROID provides an energy-aware mutation framework which is able to assess the quality of test suites in regards of their ability of unveiling energy-related defects and to improve them by uncovering missing test cases.
In order to create a proper testing framework for energy-related defects like µDROID Jabbarvand and Malek [8] had to overcome two challenges. The first challenge was the construction of a defect model which describes the creation of

the different app mutants. In order to construct this defect model an extensive list of energy anti-patterns in Android has been collected and analyzed.
The second challenge was the creation of an automatic oracle which can automatically compare the results of the mutation tests of µDROID without the developer having to compare these results manually and thus saving time and resources.

In Section 2 important concepts regarding mutation testing and Android activities are explained, which are needed in order to understand the µDROID framework. Section 3 shows an overview of the individual components of the µDROID framework and describes them in detail. An overview of related literature is given in Section 4. The final section contains the conclusion which summarizes the topics presented in this paper and shows an evaluation of µDROID for use in practice.

2 Fundamentals

This section describes some basic concepts like mutation testing (section 2.1) and the Android activity lifecycle (section 2.2) which are needed in order to understand the functionality principle of µDROID.

2.1 Mutation Testing

Mutation testing is a technique that is used to evaluate a test suite's ability to uncover defects of the program under test and discover missing test cases. This approach generates so called *mutants* from the program under test which are each seeded with a small artificial defect that developers commonly make during development. These artificial defects are called *mutation operators*. Mutation operators can either be based on a defect model which is a set of rules that represents commonly made mistakes by developers or they can be based on the syntactical mutation of elements of the programming language (e.g. replace a less-than operator by a less-than-or-equal operator) [9].

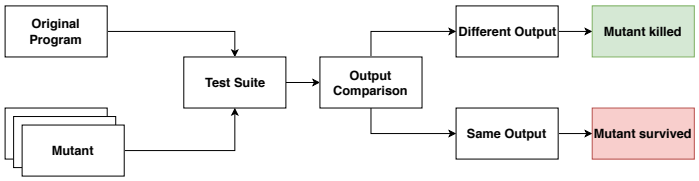

Fig. 1. Mutation testing procedure

Figure 1 describes the procedure of mutation testing. At first the test suite you want to test is executed on the original program and each mutant. Then the

outputs of the original program and the mutants get compared. If the original program and the mutant produce different outputs the mutant is detected which is also known as *killing the mutant*. If the outputs are the same the mutant is not detected and the *mutant survived*.

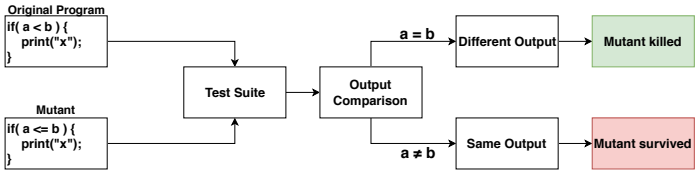

Fig. 2. Mutation testing example

Figure 2 illustrates an example. The mutant replaced the less-than operator by a less-than-or-equal operator from the original program. After executing our test suite on the original program and the mutant the outputs are compared. If the test suite tests for the case $a = b$ the original program and the mutant produce different outputs (the mutant prints an x, the original program does not). This means the mutant was killed. If the test suite does not test for the case $a = b$ the original program and the mutant produce the same output which means the introduced defect in the mutant was not detected and the mutant survived. This tells us that our test suite is missing the important test case $a = b$ and therefore should be improved.

2.2 Android Activity Lifecycle

Unlike most other programming paradigms which execute a *main()* method after starting a program and calling different functions from it, that are often used for desktop programs, an Android application consists of multiple *activites* which are each responsible for a set of specific tasks. This is due to a mobile application often having different entry points to the application depending on the context of its usage. If you open your email application directly from your home screen it shows a list of your received emails. If you click on an email address your email application is opened with the screen for writing an email [2].

Figure 3 shows the Android activity lifecycle. An activity can be in multiple states during its lifetime. After launching, the activity transitions through the green states until it is fully launched and reaches the *Resumed* state. Prior to reaching a new state a callback method is invoked. For example the onCreate() method is called before the activity transitions from the *Launch* to the *Created* state. Functions that need to be performed before presenting the activity to the user can be implemented or called in the onCreate() method. An activity can also be paused and resumed later on. After an activity is done with its task the onDestroy() method will be called and the activity will be destroyed [1].

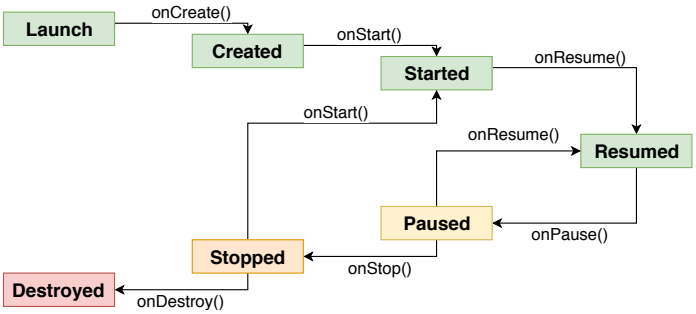

Fig. 3. Android Activity Lifecycle, based on [1]

3 µDROID Framework

Figure 4 shows the complete framework of µDROID. The procedure of µDROID starts with an *eclipse plugin* (section 3.2) which receives the source code of the original Android app and the set of mutation operators (section 3.1), that describes the rules for the mutations, as input and then generates the individual mutants out of these inputs. Then the *runner/profiler* (section 3.3) executes the test suite that should be assessed by µDROID on the original app and the generated mutants. The runner/profiler produces *power traces*, which show the energy consumption of the app over a certain amount of time, as output. These power traces are then analyzed and compared to each other by the *analysis engine* (section 3.4). Based on the results of the analysis engine a *mutation score* (section 3.5) is computed which represents the final output of the µDROID framework. The eclipse plugin and the analysis engine are executed on a desktop computer, while the runner/profiler and the test suite are executed directly on the Android device.
In the following sections each component as well as the inputs and outputs of the µDROID framework will be explained in detail.

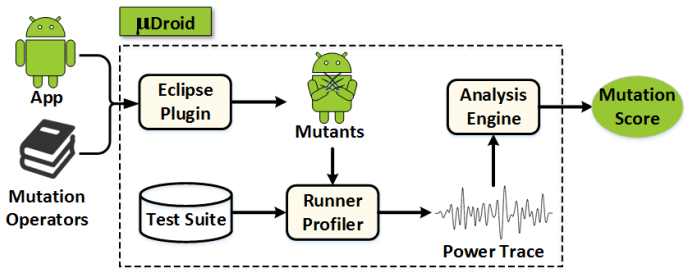

Fig. 4. µDROID framework overview [8]

3.1 Mutation Operators

The mutation operators describe how energy defects are introduced into the mutants source code. µDROID provides a total of 50 mutation operators which are divided into six classes.

Connectivity Mutation Operators
Connectivity mutation operators introduce energy defects which are related to WiFi, radio or Bluetooth usage. Figure 5 and 6 show the application of a connectivity mutation operator.
Downloading a big file using mobile data consumes a lot more energy than downloading it using WiFi. Therefore the developer should always check if WiFi is enabled before downloading a big file. The mutation operator from the example removes this check and replaces it by a true statement. If the test suite never disables WiFi and always performs its tests using WiFi instead of mobile data the power traces of the mutant and the original app will look the same and therefore the mutant survives. This behaviour can be observed in figure 5. Figure 6 illustrates the test suite disabling WiFi during testing. If WiFi is disabled the mutants power trace will be a lot bigger than the original apps power trace, therefore killing the mutant. If the mutant with this mutation operator survives all tests of the test suite it tells the developer that his test suite is missing test cases which disable WiFi usage.

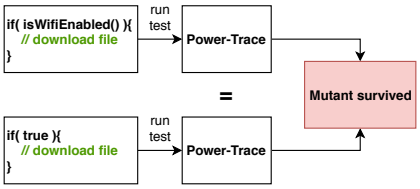

Fig. 5. WiFi always enabled in test

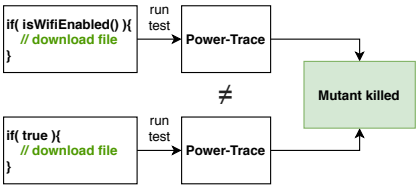

Fig. 6. WiFi disabled in test

Location Mutation Operators
Location mutation operators introduce energy defects which are commonly made during obtaining the devices location. Figure 7 illustrates the procedure to obtain a location in Android. During activity creation a LocationListener has to be registered. It then continues to update the devices location until it is unregistered. Figure 8 shows the introduction of a location mutant operator. The mutation operator removes the unregistration of the LocationListener in line 10, when the activity is destroyed. This causes the mutant to update the location of the device even though the location is not needed anymore after the destruction of the TrackActivity. In order to kill the mutant the test suite has to continue to mock locations even after the TrackActivity was already destroyed.

```
1 public class TrackActivity extends Activity{
2     private LocationManager manager;
3     private LocationListener listener;
4     protected void onCreate(){
5         manager = getSystemService("LOCATION_SERVICE");
6         // code for obtaining location
7     }
8     protected void onDestroy(){
9         super.onDestroy();
10        manager.removeUpdates(listener);
11    }
12 }
```

Fig. 7. Original location source code, based on [8]

```
1 public class TrackActivity extends Activity{
2     private LocationManager manager;
3     private LocationListener listener;
4     protected void onCreate(){
5         manager = getSystemService("LOCATION_SERVICE");
6         // code for obtaining location
7     }
8     protected void onDestroy(){
9         super.onDestroy();
10
11    }
12 }
```

Fig. 8. Location mutation operator introduced, based on [8]

Wakelock Mutation Operators
During a wakelock in Android the device or a part of it is kept awake by the app. The device can not go into an idle state during this time. In order for an app to request a wakelock the activity has to register it and after not needing it anymore to unregister it. Wakelock mutation operators are similar to location mutation operators. They remove the unregistration of the wakelock request for each component of the device.

Display Mutation Operators
The display is the most power consuming component of a device. That is why energy defects related to the display have to be thoroughly checked. Display mutation operators introduce display related energy defects like setting the screen brightness to maximum or turning off the display timeout completely.

Recurring Callback and Loop Mutation Operators
Recurring callbacks are often used to implement repeating tasks in Android. The frequency of these repeating tasks should be related to the current battery level of the device. If the energy level falls beneath a threshold like 15% the frequency should decrease accordingly. Recurring callback and loop mutation operators remove this frequency decrease. In order to kill a mutant which contains a mutation operator of this class the test suite has to perform tests with mocked battery levels.

Sensor Mutation Operators
To obtain sensor values in Android a SensorEventListener has to be registered during activity creation and unregistered once sensor values are not needed any more. Sensor mutation operators remove this unregistration similar to location mutation operators. The device keeps requesting sensor values even though the app does not need them anymore and therefore keeps consuming energy. Other mutation operators of this class increase the frequency of sensor value updates.

3.2 Eclipse Plugin

The eclipse plugin is responsible for the mutant generation. Figure 9 shows the workflow of it. First the plugin takes the source code of the original app and generates the abstract syntax tree (AST) representation of it. Then it modifies the AST according to the rules the mutation operators specify and hereby introduces energy defects into the AST representation of the source code. Finally it then transforms the AST representation back into program code. This program code represents the implementation of the mutant.

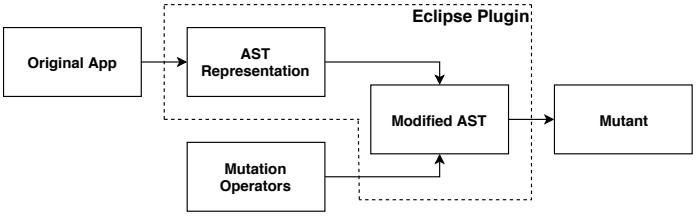

Fig. 9. µDROID eclipse plugin workflow

3.3 Runner/Profiler

The runner/profiler is responsible for executing the test suite that is to be evaluated on the original app and each generated mutant version of it. It generates a power trace for each test in the test suite which represents the energy consumption of the individual app version during test execution. Figure 10 shows an example for generated power traces. It shows power traces for an app called Sensorium, which collects several sensor values of the device it is executed on like WiFi usage or the devices current location. Power trace (a) shows the energy consumption of the execution of the original unmodified app. Power trace (b) shows the energy consumption of a mutant of the app which contains the *redundant location update mutation operator*. It continues to update its location in an unnecessarily high frequency. These redundant location updates can be observed in the power trace. The mutant version of Sensorium has an observably higher energy consumption than the original version.

3.4 Analysis Engine

Comparing the original and the mutant power traces is a tedious and time-intensive task for the developer. The µDROID framework provides an analysis engine which is able to automatically compare power traces and based on this comparison to predict whether the mutant was killed by the test suite or whether it survived. Figure 11 illustrates the workflow of the analysis engine during one

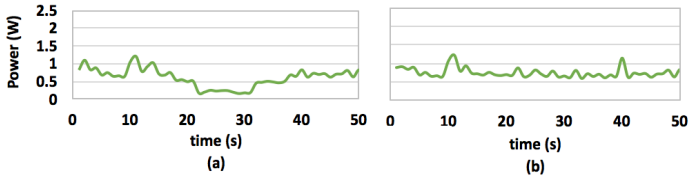

Fig. 10. Power-Traces of Sensorium app [8]

run. This procedure is executed for each test in the test suite. The analysis engine receives the power traces of the mutants and 30 power traces of the original app as input. It selects one representative power trace for the original app to account for fluctuation in the power measurement. After that the analysis engine computes the DTW distance [4] (DTW stands for *Dynamic Time Warping*. DTW is an algorithm that is used to compare the similarity between two sequences.) of the representative power trace and one mutant power trace. This DTW distance of the original and the mutant power trace then gets compared against a threshold which represents the upper bound of a 95% confidence interval which is calculated from the 30 original power traces. This means that if the DTW distance is lower than the threshold there is a 95% probability that the original and the mutant power traces are the same besides some noise. If the DTW distance is higher than the threshold the analysis engine predicts that the original and the mutant power traces differ which means that the mutant was detected and therefore killed. If it is lower than the threshold the analysis engine predicts that the mutant was not detected and therefore survived. This procedure is then repeated for each of the remaining mutants.

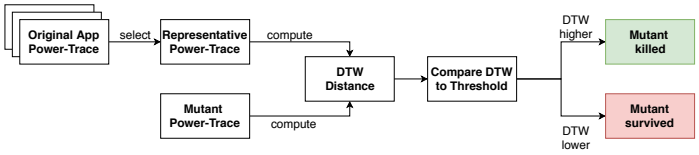

Fig. 11. Single run of the analysis engine

3.5 Mutation Score

The mutation score is the final output of µDROID. It is calculated, based on the results of the analysis engine, as follows:

$$Mutation\ Score = \frac{\#Mutants\ killed}{\#Mutants}$$

It describes the percentage of mutants that have been killed by the test suite. The higher the value, the higher the amount of mutants that were killed in relation to the amount of mutants that were generated. The mutation score indicates how good a test suite's ability of uncovering energy defects is. A high mutation score indicates a really high ability of uncovering energy defects, while a low score indicates a very bad ability of uncovering energy defects.

4 Related Work

The µDROID framework is based on a paper [7] that was also released by Jabbarvand and Malek. This paper deals with ways to advance the energy testing of mobile applications. Jabbarvand and Malek propose a three-step plan to create an energy testing framework for mobile applications. The first step is the creation of a energy-aware test generation tool that is able to automatically generate tests which exercise the energy efficiency of mobile applications. The second step is the development of a energy-aware test suite adequacy assessment tool which is capable of evaluating a test suite's ability to uncover energy defects. The third step is the creation of a energy-aware test suite minimization tool which is responsible for reducing the amount of tests that are needed to check the applications energy efficiency in order to save time and resources. µDROID is the tool which is used as the second step of their three-step plan. Besides proposing this three-step plan they also started collecting energy anti-patterns in this paper which are used to construct the different mutation operators used by µDROID.
Currently there is no further work that is based on the µDROID framework, however there is further work [6] from the authors of µDROID, Jabbarvand and Malek which continues to focus on the advancement of energy testing of mobile applications. In [6] they present the COBWEB framework which is a tool that uses a search-based algorithm to automatically generate energy tests for Android. The COBWEB framework is used as the first step of their three-step plan to advance energy testing for mobile applications.
There are several other proposed mutation testing frameworks which focus on functional correctness instead of energy efficiency. Just [10] proposes a mutation testing framework called Major which is a functional correctness testing framework for Java. Deng et al. present a mutation testing framework in [5] which is a framework for functional correctness testing for Android. They provide five classes of functional correctness mutation operators. Four of them cover Android specific app elements while the fifth covers common mistakes from Android developers.

5 Conclusion

This paper presented the energy-aware mutation testing framework µDROID. A framework which is able to assess the quality of a test suite's ability to uncover energy-related defects. Since there is currently a lack of energy testing tools

µDROID provides a service which is a real need in its domain.
In the beginning of this paper basic concepts like mutation testing and the Android activity lifecycle were explained which the knowledge of is needed in order to understand the concepts of µDROID. Afterwards an overview of µDROIDs components and their individual functionality in detail has been presented.
On µDROID conducted experiments [8] on 100 randomly selected open source apps show promising results for the usage of µDROID in practice. For each app on average 28 mutants could be generated. The amount of mutants ranged from five to 110 with each mutation operator being present at least once which indicates that even small apps can benefit from µDROID. Test suites developed by two experienced Android developers increased their mutation score from 34.5% to 90.4% after using µDROID to assess and improve their test suites which shows that even experienced developers can profit from µDROID. Additionally the analysis engine showed an accuracy of 94% in its predictions which allows for an automated application of µDROID. These results indicate that µDROID offers a promising ability for use in practice and is able to address the need for an energy-aware testing framework for Android devices.

References

[1] Android api guide, android activity lifecycle. `https://developer.android.com/guide/components/activities/activity-lifecycle`. [Online; accessed 02-December-2019].
[2] Android api guide, introduction to activities. `https://developer.android.com/guide/components/activities/intro-activities`. [Online; accessed 02-December-2019].
[3] Dr. Hannes Ametsreiter. Smartphone-markt: Konjunktur und trends. `https://www.bitkom.org/sites/default/files/2019-02/Bitkom-Pressekonferenz%20Smartphone-Markt%2020%2002%202019%20Pr%C3%A4sentation_final.pdf`, February 2019. [Online; accessed 01-December-2019].
[4] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA, 1994.
[5] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 81:154–168, 2017.
[6] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. Search-based energy testing of android. In *Proceedings of the 41st International Conference on Software Engineering*, pages 1119–1130. IEEE Press, 2019.
[7] Reyhaneh Jabbarvand and Sam Malek. Advancing energy testing of mobile applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 491–492. IEEE, 2017.
[8] Reyhaneh Jabbarvand and Sam Malek. μDROID: An Energy-Aware Mutation Testing Framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 208–219. ACM, 2017.
[9] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.
[10] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 433–436. ACM, 2014.