Automated Construction of Energy Test Oracles for Android

Reyhaneh Jabbarvand
University of Illinois at Urbana-Champaign, USA
reyhaneh@illinois.edu

Forough Mehralian and Sam Malek
University of California at Irvine, USA
{fmehrali,malek}@uci.edu

ABSTRACT

Energy efficiency is an increasingly important quality attribute for software, particularly for mobile apps. Just like any other software attribute, energy behavior of mobile apps should be properly tested prior to their release. However, mobile apps are riddled with energy defects, as currently there is a lack of proper energy testing tools. Indeed, energy testing is a fledgling area of research and recent advances have mainly focused on test input generation. This paper presents ACETON, the first approach aimed at solving the oracle problem for testing the energy behavior of mobile apps. ACETON employs Deep Learning to automatically construct an oracle that not only determines whether a test execution reveals an energy defect, but also the type of energy defect. By carefully selecting features that can be monitored on any app and mobile device, we are assured the oracle constructed using ACETON is highly reusable. Our experiments show that the oracle produced by ACETON is both highly accurate, achieving an overall precision and recall of 99%, and efficient, detecting the existence of energy defects in only 37 milliseconds on average.

CCS CONCEPTS

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

KEYWORDS

Software Testing, Test Oracle, Deep Learning, Green Software Engineering, Android

ACM Reference Format:
Reyhaneh Jabbarvand and Forough Mehralian and Sam Malek. 2020. Automated Construction of Energy Test Oracles for Android. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409677

1 INTRODUCTION

Improper usage of energy-greedy hardware components on a mobile device, such as GPS, WiFi, radio, Bluetooth, and display, can drastically discharge its battery. Recent studies have shown energy to be a major concern for both users [43] and developers [37]. In spite of that, many mobile apps abound with energy defects. This is mainly due to the lack of tools and techniques for effectively testing the energy behavior of apps prior to their release.

In fact, advancements on mobile app testing have in large part focused on functional correctness, rather than non-functional properties, such as energy efficiency [29]. To alleviate this shortcoming, recent studies have tried to generate effective energy tests [19, 28]. While the proposed techniques have shown to be effective for generating energy-aware test inputs, they either use manually constructed oracles [19, 28] or rely on observation of *power traces*, i.e., series of energy consumption measurements throughout the test execution, to determine the outcome of energy testing [9, 10, 29].

Test oracle automation is one of the most challenging facets of test automation, and in fact, has received significantly less attention in the literature [11]. A test oracle compares the output of a program under test for a given test to the output that it determines to be correct. While power trace is an important output from an energy perspective, relying on that for creating energy test oracles faces several non-trivial complications. First, collecting power traces is unwieldy, as it requires additional hardware, e.g., Monsoon [4], or specialized software, e.g., Trepn [5], to measure the power consumption of a device during test execution. Second, noise and fluctuation in power measurement may cause many tests to become flaky. Third, power trace-based oracles are device dependent, making them useless for tests intended for execution on different devices. Finally, power traces are sensitive to small changes in the code, thus are impractical for regression testing.

The key insight in our work is that whether a test fails—detects an energy defect—or passes can be determined by comparing the state of app lifecycle and hardware elements *before*, *during*, and *after* the execution of a test. If such a state changes in specific ways, we can determine that the test is failing, i.e., reveals an energy issue, irrespective of the power trace or hardware-specific differences. The challenge here lies in the fact that determining such patterns is exceptionally cumbersome, and requires deep knowledge of energy faults and their impact on the app lifecycle and hardware elements. Furthermore, energy defects change, and new types of defects emerge, as mobile platforms evolve, making it impractical to manually derive such patterns.

To overcome this challenge, we present ACETON, an approach for **a**utomated **c**onstruction of **e**nergy **t**est **o**racles for A**n**droid. ACETON employs *Deep Learning* to determine the (mis)behaviors corresponding to the different types of energy defects. It represents the state of app lifecycle and hardware elements in the form of a feature vector, called *State Vector (SV)*. Each instance of our training dataset is a sequence of SVs sampled before, during, and after the execution of a test. ACETON leverages *Attention* mechanism [8] to ensure generation of explainable DL models. This paper makes the following contributions:

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409677

- A Deep Learning technique for automated construction of an energy test oracle in Android apps that relies on a novel representation of app lifecycle and hardware elements as a feature vector. ACETON is app and device independent.
- A novel utilization of *Attention Mechanism* from the Deep Learning literature to go beyond the usage of Deep Learning as a black-box technique and understand how ACETON determines the correctness of test execution outcome.
- An extensive empirical evaluation on real-world Android apps demonstrating that ACETON is (1) highly accurate—achieves an overall precision and recall of 99%, (2) efficient—detects the existence of energy defects in only 37 milliseconds on average, and (3) reusable across a variety of apps and devices.
- An implementation of ACETON, which is publicly available [7].

The remainder of this paper is organized as follows. Section 2 provides a background on energy defects and illustrates a motivating example. Section 3 provides an overview of ACETON, while Sections 4-6 describe details of the proposed approach. Section 7 presents the evaluation results. The paper concludes with a discussion of the related research.

2 MOTIVATING EXAMPLE

An energy defect occurs when the execution of code leads to *unnecessary* energy consumption. The root cause of such issues is typically misuse of hardware elements on the mobile device by apps or Android framework under peculiar conditions. To determine whether test execution reveals an energy defect, developers can monitor the state of hardware elements and environmental factors, e.g., speed of user or strength of network signal, *before*, *during*, and *after* the test execution. If those states change in a specific way (or do not change as expected) between consecutive observations, it can be an indicator of energy defect.

For example, When developing location-aware apps, developers should use a location update strategy that achieves the proper trade-off between accuracy and energy consumption [3]. User location can be obtained by registering a `LocationListener`. While the accuracy of the location updates obtained from a *GPS* location listener is higher than that of a *Network* location listener, GPS consumes more power than Network to collect location information. To achieve the best strategy, developers should adjust the accuracy and frequency of listening to location updates based on the user movement. Example of violating the best practice is when the app uses GPS to listen to location updates while the user is stationary. This energy defect can be detected if the following pattern in the state of user and GPS hardware is observed during test execution:

$$GPS_i == GPS_{i+1} == \text{“}On\text{”} \quad \wedge$$
$$Location\_Listener_i == Location\_Listener_{i+1} == \text{“}GPS\text{”} \quad \wedge$$
$$User\_Movement_i == User\_Movement_{i+1} == \text{“}Stationary\text{”}$$

Here, GPS, $Location\_Listener$, and $User\_Movement$ are the factors corresponding to manifestation of energy defect and the indices indicate to which state, $State_i$ or $State_{i+1}$, they belong.

As shown in the above example, existence of a defect can be determined by monitoring for certain patterns in the state of hardware and environmental settings during test execution. Identifying such patterns manually requires significant expertise, and can be extremely complicated and time consuming. For example, a pattern corresponding to violation of location best practice by listening to location updates at a high frequency when the user moves slowly, i.e., walking, should include additional invariants related to $Network$ and $Listener\_Frequency$. Thereby, our objective in this paper is to construct an oracle that automatically learns such patterns to determine the correctness of test execution. Such oracle can be reusable across different apps and mobile devices, as long as the changes in the state of software and hardware can be monitored. Automatic construction of test oracles this way is specifically important for Android, as the platform rapidly evolves, i.e., substantial amounts of APIs become deprecated and new APIs and features are introduced in newer versions.

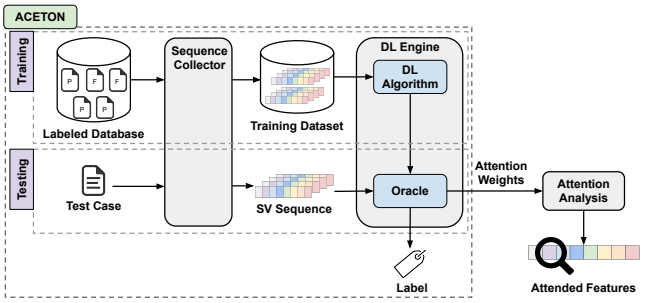

Figure 1: Overview of the ACETON framework

3 APPROACH OVERVIEW

Prior research has shown that energy defects manifest themselves under specific *contextual settings* [28]. Specifically, some energy defects, e.g., wakelocks and resource leaks, happen under specific sequences of lifecycle callbacks, while others manifest themselves under peculiar hardware states, e.g., poor network signal, no network connection, or low battery. This observation forms the basis of our work. We hypothesize an automated energy test oracle can be constructed by monitoring and comparing the state of app lifecycle and hardware elements *before*, *during*, and *after* the execution of a test. If such a state changes in specific ways, the oracle determines that the test is failing, i.e., reveals an energy issue.

Determining such patterns requires a deep knowledge of both energy defects and their corresponding impact on the app lifecycle and hardware elements. To overcome this challenge, ACETON leverages *Deep Learning (DL)* techniques to automatically *learn* the (mis)behaviors corresponding to the different types of energy defects. Specifically, ACETON monitors the state of app lifecycle and hardware elements during the execution of a test. Each sampled state is represented as a bit vector, called *State Vector (SV)*. The result of executing a test is thus a sequence of SVs, which serves as the feature vector for the DL algorithm. Each instance of training and test dataset is a sequence of SVs sampled during the execution of a test. ACETON feeds the SVs and their corresponding labels (indicating the presence of an energy defect or not) to a *Long Short Term Memory (LSTM)* network, which is a variant of *Recurrent Neural Networks (RNNs)*, to train a classifier. This classifier is subsequently used as our test oracle to determine the label of new tests.

The DL engine of ACETON uses *Attention Mechanism*, a method for making the RNNs work better by letting the network know

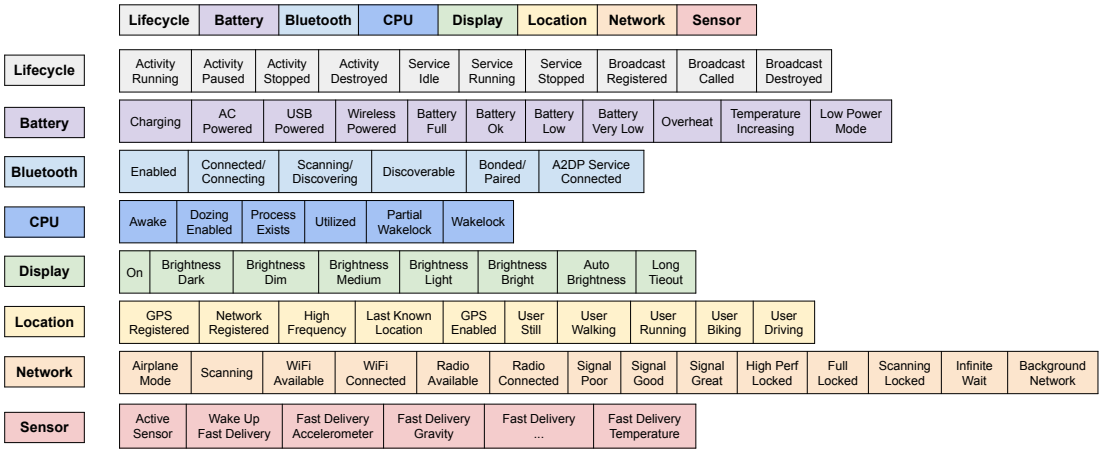

Figure 2: State Vector Representation

where to look as it predicts a label [8], to generate an explainable model. Specifically, ACETON is able to identify a subset of SVs that the oracle attends to for determining the final passing or failing outcome. By analyzing in what features the attended SVs are different from their predecessor SVs, we can verify whether the DL model has attended to the relevant features corresponding to the energy defects, in order to determine the correctness of a test.

Figure 1 provides an overview of our proposed approach, consisting of three major components: (1) *Sequence Collector*, (2) *DL Engine*, and (3) *Attention Analysis*. To construct the oracle, ACETON takes a labeled database of apps with energy defects accompanied with test suites as input. The *Sequence Collector* component executes each test case and captures SVs at a fixed rate during test execution to build the training dataset for ACETON. The training dataset is then fed to the *DL Engine*, which constructs the classifier that serves as our test oracle. To use the oracle, ACETON takes a test case as input and collects a sequence of SVs during its execution. The oracle takes the sequence as input and produces a fine-grained label for it, indicating whether the test has failed, and if so, the type of energy defect that was revealed by the test. To help us understand the nature of patterns learned by the model, the oracle also produces an *Attention Weights* vector. *Attention Analysis* component then takes the Attention Weights vector to determine the list of features that involved in the oracle's decision. These features essentially constitute the *defect signature* learned by the oracle. In the following sections, we describe the details of ACETON's components.

4 SEQUENCE COLLECTOR

The *Sequence Collector* component takes a test case t_i as input, executes it, and captures the state of app lifecycle and hardware components at a fixed rate to generate a sequence of SVs, $\overrightarrow{Seq_i} = \langle SV_0, SV_1, \ldots, SV_m \rangle$. In ACETON, $\overrightarrow{Seq_i}$ serves as the feature vector for the DL algorithm. In this section, we first explain details of SV and then describe the process of sequence collection.

4.1 State Vector (SV)

Proper feature selection, i.e., feature engineering, is fundamental to the application of DL techniques, as the quality and quantity of features greatly impact the utility of a model. We chose our features to reflect the changes in the state of app lifecycle and hardware elements during the execution of a test, as these factors have shown to play an important role in manifestation of energy defects [28]. To capture the state during the execution of a test, ACETON relies on a model called *State Vector (SV)*. At the highest level, SV consists of entries representing the lifecycle state of app under test and the state of major energy-greedy hardware elements, namely Battery, Bluetooth, CPU, Display, Location, Network (e.g., WiFi or radio), and Sensors (e.g., Accelerometer, Gravity, Gyroscope, Temperature, etc.), $\overrightarrow{SV} = \langle C_0, C_1, \ldots, C_7 \rangle$, where C represents the element category. At a finer granularity, each category is broken down to sub-entries that capture the corresponding state in terms of multiple features, $\overrightarrow{C_j} = \langle f_0, f_1, \ldots, f_{n_j} \rangle$, where f is a binary value representing the state of feature.

Figure 2 demonstrates the representation of SV at the highest level in the first row and at a finer granularity for all the entries. As shown in Figure 2, Location element consists of ten sub-entries, namely *GPS Registered* (indicates whether a GPS listener is registered by an app), *Network Registered* (indicates whether a Network listener is registered by an app), *High Frequency* (indicates if the registered location listener listens to location updates frequently), *Last Known Location* (indicates whether the last known location is available for an app), *GPS Enabled* (indicates whether the GPS hardware is on or off), and entries indicating the type of user movement as the test executes.

To determine sub-entries, i.e., features, we needed two sets of information: (1) a set of lifecycle states for Android components, i.e., Activity, Lifecycle, and BroadcastReceiver, and (2) states of key hardware elements that can be changed at the software level. We referred to Android documentation [1, 2, 6] to determine the former. For the latter, we followed a systematic approach similar to that presented in the prior work [28] to obtain all the Android APIs and constant values in the libraries that allow developers to monitor or utilize hardware components. Specifically, we performed a keyword-based search on the Android API documentation to collect hardware-relevant APIs and fields, identified all the hardware states that can be changed or monitored at the software level, and constructed State Vector as demonstrated in Figure 2. By identifying

the hardware features using the mentioned approach, i.e., determining the hardware states that can be manipulated or monitored using application software or Android framework, we are assured the oracles constructed following our approach are *device independent*. Additionally, the constructed oracle is *app independent*, as it monitors the features that are related to app's lifecycle state, which are managed by Android framework, in contrast to features that are related to the code of apps, e.g., usage of specific APIs. Thereby, once trained on a set of apps, the oracle can be reused for testing of other apps.

An SV consists of a total of 84 binary features. We leveraged One-Hot encoding to transform all the categorical data into binary values. For example, while user movement can be a single feature with categorical text values of *Still*, *Walking*, *Running*, *Biking*, and *Driving*[1], we model it as five binary features. This is mainly because binary features are easier to learn by DL techniques, thereby leading to a higher level of accuracy in a shorter amount of time.

4.2 Collecting Sequences

The *Sequence Collector* component executes a given test, t_i, and collects the values for different sub-entries of SV at a fixed sampling rate to generate $\overrightarrow{Seq_i}$. ACETON's *DL Engine* requires the size of all the $\overrightarrow{Seq_i}$s be the same. Since tests may take different amounts of time to execute, *Sequence Collector* adjusts the frequency of sampling based on the length of tests. Current implementation of ACETON requires 128 SV samples (details in Section 7).

Sequence Collector leverages *dumpsys* and *systrace* capabilities of the *Android Debug Bridge (ADB)*, along with instrumentation of apps, to collect the necessary information at different time stamps. *dumpsys* is a command-line tool that provides information about system services, such as *batterystats*, *connectivity*, *wifi*, *power*, etc. For example, "`adb shell dumpsys wifi`" command collects and dumps the statistics related to the WiFi hardware element. To determine if there is a WiFi network available, we look at the value of "`WiFi is`" line in the dumpsys report. Similarly, to see if the phone is connected to a WiFi network, we look at the value of "`curState`". If "`curState = NotConnectedState`", the phone is not connected to a WiFi network. If "`curState = ConnectedState`", the phone has connection to a WiFi network, in which case we collect additional information about the connection, e.g., the strength of the signal.

While *dumpsys* provides detailed information about all the running services on a phone, its reporting time for CPU is very long. That is, it batches all the CPU usage information and updates the CPU report every several minutes. Thereby, we used *systrace* to collect the information about CPU usage of an app during test execution. Finally, we could not find information in either *dumpsys* or *systrace* report for a subset of features. To that end, ACETON automatically instruments the app under test to collect such information. For example, *Location* category contains features related to the type of user movement. To identify how and when user movement changes, ACETON instruments the app to register an *Activity Recognition* listener and listens to the changes in user movement. That is, when the device recognizes a change in the user movement by collecting the data from various sensors, Android will notify the listener about the type of detected activity, e.g., walking, running. As another example, all the lifecycle callbacks will be instrumented to print a message Android log, i.e., LogCat, as they are invoked. By processing the log files collected for an SV during test execution, we determine the values for lifecycle features.

[1]These categories are specified in the Android documentation.

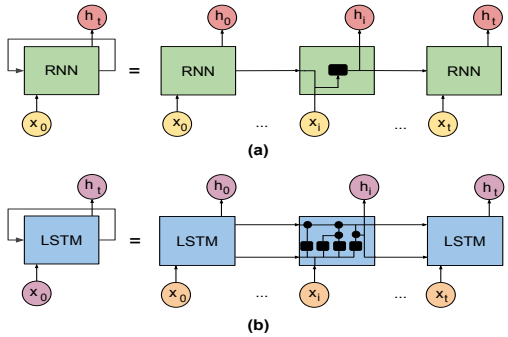

Figure 3: Architecture of an RNN and LSTM networks

5 LEARNING ENGINE

In this section, we describe the DL-based construction of our energy test oracle.

5.1 Model Selection

To determine what Machine Learning model is suitable for solving the energy oracle problem, we considered the following criteria:

1) The construction of energy oracle is a form of *Classification* problem, i.e., we train our model based on a set of labeled passing or failing tests. Hence, the model should be suitable for such supervised learning problem;

2) We have a relatively high-dimensional data, i.e., each single input to the model is a sequence of SVs sampled during execution of a test. For a sequence size of 128 and SV size of 84 with binary features, each instance of our feature vector can take $128 \times 84 = 2,1504$ values. Thereby, the model should be able to deal with both *sequential* and *high-dimensional* data;

3) Energy defects can occur anywhere during the execution of a test. As a result, the index of SVs where an energy defect occurs can be different among tests. Thereby, our proposed oracle should be able to detect emergence of the anomalous energy behavior in SV_k from the SVs that appear before it, $\{SV_l \mid l < k\}$. That is, our model should be able to holistically consider the observed SVs in order to accurately detect energy defects.

Given these criteria, the learning component of ACETON uses *Long Short-Term Memory (LSTM)*, which is a type of *Recurrent Neural Network (RNN)*. Specifically, ACETON uses an LSTM Neural Network, augmented by *Attention* mechanism, to construct oracles for energy defects in Android. In the remainder of this section, we describe the intuition behind why LSTM is the best DL model for construction of an energy test oracle.

5.2 Long Short-Term Memory (LSTM)

Neural Networks (NNs) have been widely used to recognize underlying relationships in a set of data through a statistical process.

Such systems learn to perform a task or predict an output by considering examples (supervised learning) rather than pre-defined rules. For example, NN algorithms have been shown to effectively identify presence of a certain object in a given image, only by analyzing previously seen images that contain that object and without knowing its particular properties. Neural Networks are basically a collection of nodes, i.e., artificial neurons, which are typically aggregated into layers. The network forms by connecting the output of certain neurons in one layer to the input of other neurons in the predecessor layer, forming a directed, weighted graph. Neurons and their corresponding edges typically have a weight that adjusts as the learning proceeds. "Classic" NNs transmit information between neurons in a single direction, thereby are not effective in dealing with sequential data.

Recurrent Neural Networks (RNNs) are specific type of NNs that have shown to be effective in solving large and complex problems with sequential data, e.g., speech recognition, translation, and time-series forecasting. They are networks with loops in them, which allows them to read the input data one sequence after the other. That is, if the input data consists of a sequence of length k, RNN reads the data in a loop with k iterations. Figure 3-a shows the architecture of an RNN on the left, which is unfolded over time on the right. While the chain-like nature of RNNs enables them to reason about previous sequences, basic RNNs are unable to learn long-term dependencies due to the Vanishing Gradient problem [13].

Learning long-term dependencies is essential in the energy oracle problem, defect patterns should persist for some time in order to be considered a defect. For example, registering a GPS listener that listens to location updates as frequently as possible—by setting the time and distance parameters of `requestLocationUpdates()` to 0—is an example of an energy defect [29]. The pattern of this defect may involve *GPS Registered* and *High Frequency* sub-entries in the SV (Figure 2), i.e., turn their corresponding value to "1" as an app registers the listener. However, simply observing that pattern in a sampled SV does not necessarily entail an energy defect. That is, if developer registers a high frequency location listener in a short-lived Broadcast Receiver or Service, or set a short timeout to unregister it, the pattern does not impact the battery life, hence, should not be considered a defect. In other words, the pattern should persist among several consecutive SVs during the execution of a test, or persist after the test terminates to be an energy defect.

LSTM networks are special kind of RNNs that are capable of learning long-term dependencies [26], thereby can remember the patterns that will persist. Similar to classic RNNs, LSTMs have the form of a chain of repeating modules of neural network, as shown in Figure 3-b. However, the repeating module in LSTM (right hand side of Figure 3-b) has a different structure compared to that of RNN (right hand side of Figure 3-a). While RNNs have a single NN layer (demonstrated by black rectangle), LSTMs have four of them, which are interacting in a special way to create an internal memory state. The combination of layers enable LSTM to decide what information to throw away and what to keep, i.e., empowering LSTM to remember what it has learned till present.

The LSTM layer consists of several LSTM modules that take a sequence of SVs as input and generate an output vector, $\overrightarrow{h_m}$. A regular classification algorithm projects this output to the classification space, with dimensions equal to the number of classes, and then applies a probabilistic function, a.k.a. *softmax*, to normalize the values between $[0, 1]$ and generate a label. However, to produce more accurate labels, ACETON takes $\overrightarrow{h_m}$ as an input to an additional layer, i.e., *Attention* layer, as discussed next.

5.3 Dataset Curation

A DL approach requires the availability of large amounts of high quality training data, i.e., a large dataset with diverse types of energy defects in mobile apps accompanied by test suites that reveal their existence. We present a novel usage of mutation testing to curate such dataset. Specifically, we used μDroid, an energy-aware mutation testing framework designed for Android [29]. The rationale behind this choice includes:

(1) μDroid can provide us with a large, diverse, and high quality dataset. The mutation operators of μDroid are designed based on the most comprehensive energy defect model for Android to date, which is constructed from real energy defects obtained from several dozens of Android apps. These defects have been shown to strongly associate with previously unknown real energy defects in apps that were different from those where the defect model was derived from. μDroid also comes with a set of high quality developer-written passing and failing tests, which are essential for generating a labeled dataset for our classification problem. Each pair of $\langle mutant, test \rangle$ from μDroid contributes one data point for our dataset.

(2) μDroid categorizes mutants based on the hardware components that they misuse, providing us with fine-grained labels for failing tests, namely $Pass$, $Fail_{Bluetooth}$, $Fail_{CPU}$, $Fail_{Display}$, $Fail_{Location}$, $Fail_{Network}$, and $Fail_{Sensor}$, to perform additional analysis and verify the validity of the DL model (see Section 6).

5.4 Attention Mechanism

While LSTMs have memory, their performance drastically degrades as the length of sequences gets longer, known as the *long sequence problem* in the literature [18]. Attention mechanism is a method for making LSTMs overcome this challenge by reminding the network where it has previously looked as it performs its task [8]. Thereby, no matter how long the sequences, LSTM knows where it has focused and decides what to do next based on that information. In addition to solving the long sequence problem, Attention mechanism is extensively used in the deep learning community to resolve the explainability of neural networks.

The responsibility of Attention layer is to generate an *Attention Weight* vector, $\overrightarrow{AW} = \langle w_0, w_1, \ldots, x_m \rangle$, and adjust the weights as SVs are sequentially being fed to the LSTMs. Once the oracle receives all the SVs, $\overrightarrow{AW}$ contains weight values corresponding to each SV. ACETON uses *soft attention*, where w_i values in $\overrightarrow{AW}$ are between 0 and 1 and $\sum_{i=0}^{m} w_i = 1$. Thereby, it provides a convenient probabilistic interpretation of which SVs in the test case the oracle has relied on to determine the outcome of a given test. For example, if ACETON decides a test fails due to a location-related energy defect, i.e., predicts $Fail_{Location}$ label for it, we expect that the highest weights in $\overrightarrow{AW}$ belong to SVs in which Location sub-entries were actively changed as the test executed. If so, the model proves to focus on relevant sequences to predict the outcome. Otherwise, it has learned an incorrect pattern and might be invalid.

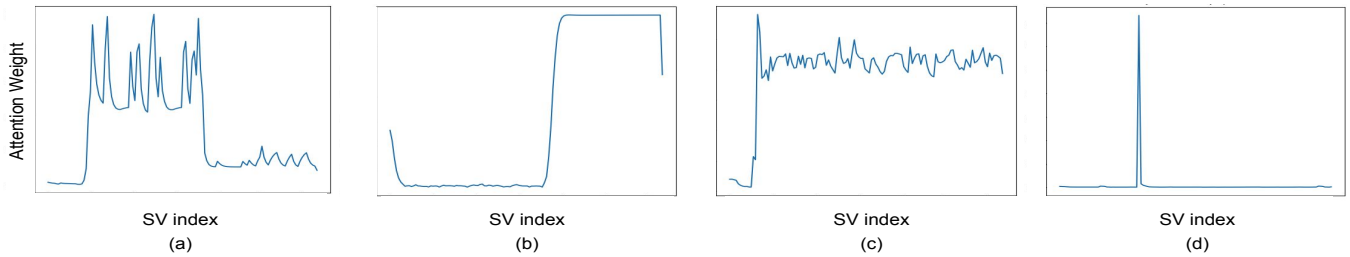

Figure 4: Visualization of Attention Weight vector for energy defects related to a) CPU, b) Display, c) Location, and d) Network

6 ATTENTION ANALYSIS

Interpretability of DL models is essential, as they are highly vulnerable to the *data leakage problem* [32]. Data leakage causes a model to create unrealistically good predictions based on learning from irrelevant features. A famous example of data leakage is a cancer predictive model that makes its decision based on the lab's label on the X-Ray, rather than focusing on the content of X-Ray itself. While this model may make good predictions, it is invalid. To ensure validity of a model, it is hence crucial to determine the features that impact its decision and verify they are relevant.

Utilization of *Attention* by itself improves the performance and accuracy of the energy oracle. ACETON takes advantage of *Attention* layer's product, i.e., *Attention Weight* vector to identify a set of features that ACETON's model has focused on to predict a label. This set can be used for two purposes: (1) verify validity of the learned model, and (2) enhance energy fault localization.

Algorithm 1 presents ACETON's approach for Attention Analysis. For a given failing test, t_i, it takes the sequence of SVs, $\overrightarrow{Seq_i} = \langle SV_0, SV_1, \ldots, SV_m \rangle$, Attention Weight vector, $\overrightarrow{AW_i}$, and predicted label, l_i, as input, and produces a list of features that were involved in the decision, i.e., *attended features*, as output. The algorithm starts by identifying a subset of SVs in $\overrightarrow{Seq_i}$ that the oracle has attended to decide the label, $\overrightarrow{Seq'_i} = \langle SV_n, , SV_k \rangle$, $0 < n \leq k < m$ (Line 2), and determines the features that are common between SVs in $\overrightarrow{Seq'_i}$ to construct $Commons_i$ (Line 3). Next, the Algorithm takes the predecessor to the first element in $\overrightarrow{Seq'_i}$, $Pred_i = SV_{n-1}$ (Line 4), and compares the values of features in $Commons_i$ with that of in $Pred_i$'s features to identify attended features, $Features_i$ (Lines 5-8).

Finally, Algorithm 1 extracts the SV category corresponding to the attended features, c_i (Line 9). If l_i matches the attended category, c_i, Algorithm 1 verifies that the model attended to the features relevant to the type of defect and returns $Features_i$ (Lines 10-11). Otherwise, it returns an empty set, as the model has attended to the incorrect SVs and might be invalid (Lines 12-13).

To explain the intuition behind Algorithm 1, consider Figure 4, which visualizes $\overrightarrow{AW}$ for four samples of our dataset, related to energy defects that engage CPU, Display, Location, and Network. Figure 4-a is for an energy defect related to the CPU, which utilizes CPU when the app is paused, i.e., goes in the background. In this example, the spike in the attention weights that remains for some time corresponds to when the test puts an app in the background. Figure 4-b is for an energy defect related to the Display that increases the display brightness to the max during app execution. The spike in this Figure is where the app increases the screen brightness by setting the screen flag. As the app terminates,

Algorithm 1: Attention Analysis Algorithm

Input: SV sequence of a failing test $\overrightarrow{Seq_i}$, Predicted label l_i, Attention Weight vector $\overrightarrow{AW_i}$
Output: Attended Features $Features_i$

```
1  Features_i ← ∅
2  Seq'_i ← getAttendedSVs(Seq_i, AW_i)
3  Commons_i = getCommonAttendedFeatures(Seq'_i)
4  Pred_i ← getPredecessor(Seq'_i)
5  foreach ⟨f_x, v_x⟩ ∈ Commons_i do
6      v'_x ← getFeatureValue(f_x, Pred_i)
7      if v'_x ≠ v_x then
8          Features_i ← Features_i ∪ f_x
9  c_i ← getAttendedCategory(Features_i)
10 if c_i matches l_i then
11     return Features_i
12 else
13     return ∅
```

Android clears the flag and the brightness goes back to normal, thereby, the attention of the model also fades. Figure 4-c is for an energy defect related to the Location, where the developer registers a listener for receiving location updates with high frequency and forgets to unregister the listener when the app terminates. In this case, attention of the model goes up at the SV index in which the app registers the listener and does not drop even when the test terminates. Finally, Figure 4-d is for a Network energy defect, where the app fails to check for connectivity before performing a network task. When there is no network connection available, the app still performs a signal search, which consumes an unnecessary battery consumption. In Figure 4-d, the attention of model lasts shorter compared to other examples, as searching for the signal is effective for a short period of time, compared to the length of test. Thereby, it appears in few sampled SVs.

As shown in Figure 4, depending on where the energy defects in these energy-greedy apps occur, how much they last, and whether their impact remains when a test terminates or not, attention of the model to the sampled SVs varies. However, there is one pattern common among them. There is always a sharp jump in the attention weights, which indicates where the model starts to notice the pattern. The spike of attention either remains until end or sharply drops after some time. To that end, Algorithm 1 sets SV_n as the start of the *biggest jump* in the weights in $\overrightarrow{AW_i}$, and SV_k as the end of *biggest drop* following the sharpest jump. If there is no sharp drop until the end of $\overrightarrow{AW_i}$, Algorithm 1 sets SV_k to the last SV in Seq_i, i.e., SV_m. The SVs between SV_n and SV_k construct Seq'_i.

The next step after identifying the attended SVs is to determine the attended features. To that end, Algorithm 1 first collects the features that are common (i.e., have the same value) among all SVs in $\overrightarrow{Seq_i'}$ to construct $Commons_i$. Formally speaking, $Commons_i := \{ \langle f_x, v_x \rangle \mid \forall SV_j \in \overrightarrow{Seq_i'},\ f_x.v_x = 1 \vee f_x.v_x = 0 \}$. That is, $Commons_i$ is a set of pairs $\langle f_x, v_x \rangle$, where the value v_x of each feature f_x among all the SVs in $\overrightarrow{Seq_i'}$ is always 0 or always 1. While these features are common among the attended SVs, not all of them are relevant to the final decision of the oracle. For example, $Commons_i$ is very likely to contain *Display On* feature in most cases, as test execution happens when the display is on. However, this feature should not appear in the attended features if a test that fails due to a Network misuse.

To exclude the irrelevant features, Algorithm 1 refers to SV_{n-1}, which is the predecessor to the first SV in $\overrightarrow{Seq_i'}$. The intuition here is that $SV_n \in \overrightarrow{Seq_i'}$ is where the model starts to attend, indicating a change in the state of lifecycle and hardware elements that cause the energy defect. Hence, SV_{n-1} indicates a safe state with no energy defect. For each f_x in $Commons_i$, Algorithm 1 finds the value of its corresponding feature in SV_{n-1}. If that value is different from v_x, Algorithm 1 adds it to the attended features $Features_i$.

Once the list of attended features is extracted, Algorithm 1 identifies the category corresponding to those features by referring to the high-level structure of SV (recall Figure 2). For example, if $Features_i$ contains *Enabled*, *Connected/Connecting*, and *Bonded/-Paired* features, category c_i is set to *Bluetooth*. If the predicted category for the given test, l_i, matches c_i, we determine that the model has attended to the right features to decide the label.

Attended features can be viewed as the footprint of energy defects on the app's lifecycle and hardware states, i.e., *Defect Signature*. Thereby, in addition to verifying the validity of the oracle, they can be used by developers to enhance the fault localization process. In fact, knowing the fine-grained properties of the app lifecycle and hardware elements that are involved in the manifestation of an energy defect can focus the developers effort on parts of the code that utilizes Android APIs related to them, making the identification of the root cause easier. For example, if the defect signature contains *GPS Registered* and *High Frequency* features from the Location category, developers are provided with strong hints that parts of the program that register location listeners for GPS and adjust the frequency of receiving location updates are culpable for the energy defect.

7 EVALUATION

We investigate the following five research questions in the evaluation of ACETON:

- **RQ1.** *Effectiveness*: How effective is the generated test oracle for detection of energy defects in Android apps?
- **RQ2.** *Usage of Attention Mechanism*: To what extent usage of Attention Mechanism improves the performance of the model? What features impact the oracle's decision?
- **RQ3.** *Detection of Unseen Energy Defects*: To what extent can ACETON detect unseen energy defect types, i.e., those that are not in the training dataset?
- **RQ4.** *Reusability of the Oracle*: Can the generated oracle be used to detect energy issues on different apps and mobile devices?

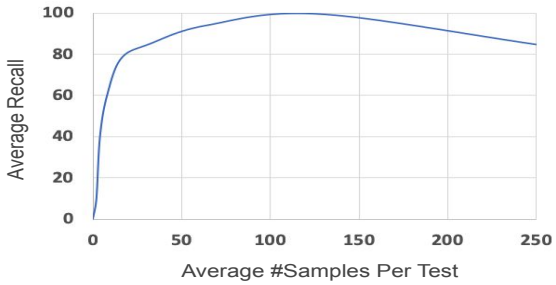

Figure 5: Sensitivity of the oracle's accuracy to sampling rate

- **RQ5.** *Performance*: How long does it take for ACETON to train and test a model?

7.1 Experimental Setup

Dataset: μDroid dataset contains 413 mutants from various categories of energy defects and comes with 329 high quality tests generated by Android developers, making it suitable to generate our dataset. Each pair of $\langle mutant, test \rangle$ from μDroid serves as a data point in our Labeled Database (Figure 1). μDroid provides only *passed* or *killed* labels for its tests. We transformed the *killed* label into a more fine-grained label in our approach (ref. Section 5.3), based on the high-level categories related to the hardware components that the mutants misuse. That is, if the killed mutant belongs to Bluetooth category in μDroid, we change its label to $Fail_{Bluetooth}$. In addition, we removed the mutants that were reported as equivalent by μDroid, as well as mutants which could not be killed by test suites, leaving us with 295 mutants containing 22 types of energy defect. The first six columns of Table 1 show details about the properties of the *Labeled Database*. Overall, the *Labeled Dataset* contains 16,347 instances of $\langle mutant, test \rangle$, where 9,266 of them are passing and 7,081 are failing.[2] We executed each instance using *Sequence Collector* component and collected corresponding SVs for each instance to generate our final dataset. Table 1 shows the details of μDroid's dataset.

DL Engine Configuration: We implemented our learning model using PyTorch [39], an open-source ML library for Python. There are multiple parameters in the implementation that impact the performance of a DL model. One of them is the loss function, which determines how well the algorithm approaches to learn a model. While *Cross-Entropy* is the most commonly used *loss function* for classification problems [47], it was not the best option in this problem due to the imbalanced nature of our dataset, i.e., the number of passing instances in our database is higher than failing ones. Thereby, we used *Weighted Cross-Entropy* [34] loss function to enforce model focus on minority classes. To enhance the performance, we utilize *Adam optimizer* [33] to update the network weights and minimize this loss function iteratively. Overfitting can also have a negative impact on the performance of a model. To overcome *Overfitting* and ensure the generalization of the model on new data, we use *early stopping technique* [40]. That is, we track the performance of the trained model on the validation dataset at each *epoch* and

[2] The actual size of Labeled Dataset in the context of DL is $1,961,640 = 16,347 \times 120$, as each $\langle mutant, test \rangle$ consists of 120 SVs, where the model should consider each of them to generate a correct label. For the sake of simplicity, we only report the size of $\langle mutant, test \rangle$ pairs.

Table 1: Properties of Labeled Database, learned defect signatures, and ACETON's performance on unseen defects.

Hardware Category	Subcategory ID	Defect Description	#Mutants	#Instances		Defect Signature	Unseen Recall
				Failing	Passing		
Bluetooth	B1	Unnecessary active Bluetooth connections	5	83	110	BE = 0, BC = 1, (AP ∨ AD ∨ SS) = 1	93.04
	B2	Frequently scan for discoverable device				BS = 1, BTI = 1	82.54
	B3	Keep discovering for devices when not interacting				BE = 1, BS = 1, AP = 1	83.33
CPU	C1	High CPU utilization	51	1704	2022	CPUA = 1, PE = 1, CPUU = 1, Charging = 0, BTI = 1, BO = 1, (AP ∨ AD ∨ SR) = 1	99.64
	C2	High CPU utilization when battery is low				CPUA = 1, PE = 1, CPUU = 1, Charging = 0, BVL = 1, BTI = 1, BO= 1 (AP ∨ AD ∨ SR) = 1	99.12
	C3	High CPU utilization when not interacting				CPUA = 1, PE = 0, CPUU = 1, Charging = 0, BTI = 1, BO = 1, (AP ∨ AD ∨ SR) = 1	98.85
	C4	Active CPU wakelock while not interacting				AD = 1, CPUW = 1	6.7*
Display	D1	Failing to restore long screen timeout	90	1506	2458	DLT = 1, (AP ∨ AD) = 1	-
	D2	Maximum screen brightness set by app				DSBB = 1, AR = 1	-
Location	L1	High frequency Location update	91	2632	3195	(GL ∨ NL) = 1, HFLU = 1, GO = 1, LKLA = 1, (US ∨ UW) = 1	97.62
	L2	Unnecessary accurate Location Listener				GL = 1, NL = 1, LKLA = 1, GO = 1, (US ∨ UW) = 1, UD = 0	99.53
	L3	Active GPS when not interacting				(GL ∨ NL) = 1, LKLA = 1, GO = 1, (AP ∨ AD) = 1, UD = 0	82.01
	L4	Neglecting Last Known Location				GL = 1, LKLA= 1, HFLU = 1, GO = 1, UD = 0	100
Network	N1	Fail to check for connectivity	46	824	1321	WS = 1, WA = 0, WC = 0	5.21*
	N2	Frequently scan for WiFi				WS = 1, WC = 1, BTI = 0, AP = 1	100
	N3	Scanning for WiFi while not interacting				WS = 1, WA = 1, (AP ∨ AD) = 1	33.33*
	N4	Using cellular over WiFi is available				WA = 1, WC = 0, RA = 1, RC = 1, (SGo ∨ SGr) = 1	97.37
	N5	Long Timeout for Corrupted Connection				WA = 1, WC = 1, ICW = 1, (AP ∨ AD) = 1	96.15
	N6	Active WiFi wakelock while not interacting				WA = 1, WC = 1, NAB = 1, (WLS ∨ WLHP) = 1, (AP ∨ AD) = 1	98.08
	N7	Improper High Performance WiFilock				WA = 1, WC = 1, SP = 1, WLHP = 1, (AP ∨ AD) = 1	100
Sensor	S1	Unnecessary active sensors	12	332	160	SA = 1, (AP ∨ AD = 1)	96.47
	S2	Fast delivery wakeup sensors				SA = 1, WFDS = 1, ASAcc = 1, ASPre = 1, ASMag = 1	94.07
Total	-	-	295	7081	9266	-	-

Table Legend:
AD: Activity Destroyed, AP: Activity Paused, AR: Activity Running, BE: Bluetooth Enabled, BC: Bluetooth Connected, BS: Bluetooth Scanning, BTI: Battery Temperature Increasing, BO: Battery Overheat, BVL: Battery Very Low, CPUA: CPU Awake, CPUU: CPU Utilized, CPUW: CPU Wakelock, DLT: Display Long Timeout, DSBB: Display Screen Brightness Bright, GL: GPS Listener, HFLU: High Frequency Location Update, GO: GPS On, LKLA: Last Known Location Available, LCT: Long Connection Timeout, NL: Network Listener, NAB: Network Active Background, PE: Process Exists, RA: Radio Available, RC: Radio Connected, SGo: Signal Good, SGr: Signal Great, SP: Signal Poor, SA: Sensor Active, SS: Service Stopped, WA: WiFi Available, WC: WiFi Connected, WLS: Wakelock Scanning, WLHP: Wakelock High Performance, WS: WiFi Scanning, UD: User Driving, US: User Still, UW: User Walking, WFDS: Wakeup Fast Delivery Sensor, ASAcc = Active Accelerometer Sensor, SPre: Active Pressure Sensor, ASMag: Active Magnetic Sensor

stop the training if there is an increasing trend in the validation loss in 2 consecutive epochs. Thereby, we get a model with the least validation loss. We have also followed the 10-fold cross validation methodology in evaluating the performance of oracle.

For *hyperparameter tuning*, we conducted a guided grid search strategy to find a configuration for the model that results in the best performance on the validation data. One of the important hyperparameters in energy oracle model is the size of sequences. To illustrate how this hyperparameter impacts performance of the oracle, consider Figure 5, which depicts the sensitivity of the energy oracle's accuracy to the average number of samples per test. As shown in this Figure, accuracy of the oracle is quite low, 61%, when we sample SVs only *before* and *after* execution of a test (*Sample Per Test = 2*). That is because a subset of energy defects, e.g., using light background, fast delivery sensor listener, and etc., happen during the execution of a test and their impact disappears when the test terminates. Therefore, our approach is unable to learn and later predict such types of energy issues with extremely low sample rates. While increasing the number of samples per test alleviates this problem, exceeding certain threshold (past 130 samples per test in Figure 5) appears to unnecessarily increase the complexity of DL problem, thereby reducing the accuracy of classifier. Other detailed configuration of DL Engine are available on ACETON's website [7].

Table 2: Comparing ability of ACETON in detecting the category of different energy defects (* indicates the wrong predictions)

	ACETON with Attention							ACETON without Attention						
	Pass	Bluetooth	CPU	Display	Location	Network	Sensor	Pass	Bluetooth	CPU	Display	Location	Network	Sensor
Pass	916	0	0	0	3*	0	0	903	0	1*	2*	3*	9*	1*
Bluetooth	0	8	0	0	0	0	0	4*	8	0	0	0	0	0
CPU	0	0	168	0	0	0	0	0	0	167	0	0	0	0
Display	0	0	0	150	0	0	0	0	0	0	148	0	0	0
Location	0	0	0	0	258	0	0	1*	0	0	0	258	0	0
Network	0	0	0	0	0	80	0	0	0	0	0	0	71	0
Sensor	0	0	0	0	0	0	32	8	0	0	0	0	0	31
Precision(%)	99.67	100	100	100	100	100	100	98.26	66.67	100	100	99.61	100	79.49
Recall(%)	100	100	100	100	98.85	100	100	98.58	100	99.4	98.67	98.85	88.75	96.88

Table 3: ACETON's performance on detection of real defects.

Apps	a2dp.Vol			Gtalk			Openbmap			Open Camera	Sensorium			Ushahidi
Version	8624c4f	8231d4d	4767d64	dce8b85	c0f8fa2	5ce2d94	56c3a67	14d166f	f72421f	1.0	e153fdf	94c9a8d	94c9a8d	4f20612
Defect Type	Location	Location	Bluetooth	CPU	Location	CPU	CPU	CPU	Network	Display	CPU	CPU	CPU	Location
Label	Location	Location	Bluetooth	CPU	Location	CPU	CPU	CPU	Network	Display	CPU	CPU	CPU	Location

7.2 RQ1: Effectiveness

While ACETON builds on top of a high-quality dataset, we performed two experiments to ensure generalizability of our results in evaluating the ability of ACETON to detect energy defects. In the first experiment, we used the *Labeled Dataset* for both training and testing purposes. In the second experiment, we trained the oracle based on the *Labeled Dataset* and used real energy defects (non-mutant apps with energy defects confirmed by their developers) to test the oracle.

7.2.1 Effectiveness on detecting mutant defects. For the purpose of this evaluation, we divided the dataset obtained from *Labeled Database* into two categories of training set, to train the oracle with it, and test set, to test the performance of oracle. That is, we downsampled each category of mutants, e.g., Location, by 90% for training, and used the remaining 10% for testing. While our feature vector is designed to reflect information that is app independent—not dependent to usage of specific APIs or code constructs—we ensured that during downsampling, the mutants in the test set belong to different apps compared to that used in the training set. This strategy accounts for overfitting and potential bias in favor of specific apps. We select *Precision* and *Recall*, and not *Accuracy*, as metrics to measure effectiveness of ACETON in predicting correct labels, since our data is imbalanced. With imbalanced classes, it is easy to get a high accuracy without actually making useful predictions, as the majority class impacts *true negative* values. Table 2 shows the result for this experiment under *ACETON with Attention* column. These results are obtained through a 10-fold cross validation, i.e., downsampling repeated 10 times.

Each row in this Table shows the number of test instances in a predicted class, where each column indicates the instances in actual class. From this result we observe that: **ACETON predicts correct labels for each category with a very high precision and recall**. In fact, ACETON was able to detect all the defects related to the *Sensor*, *Network*, *Display*, *CPU*, and *Bluetooth* and only missed 3 Location defects (marked by * in Table 2), i.e., identified them as passed. The average precision and recall values over all categories are 99.9% and 99.8%, respectively. Categorical precision and recall values are listed in the last two rows.

7.2.2 Effectiveness on detecting real defects. While ACETON is able to effectively detect the outcome of tests in mutants, we also wanted to see how it performs on Android apps that have real but similar energy defects. To that end, we referred to a prior work [28], which provides a dataset of 14 Android apps with real energy defects. Each app is accompanied by a test generated using their test generation tool, which is manually confirmed to reproduce the energy defect. The supplementary information in the artifact of that dataset also indicates the type of hardware element that is misused by the defect, which we used to identify if ACETON correctly identifies the outcome of tests. Table 3 represents the results for this experiment. As shown in Table 3, ACETON was able to correctly identify the outcome of tests on all subjects. This observation indicates that **ACETON can effectively detect real energy defects in mobile apps.**

7.3 RQ2: Usage of Attention Mechanism

Recall that we use the *Attention* mechanism for two purposes: (1) to enhance performance of the model; and (2) to verify validity of the model. In this research question, we evaluate to what extent *Attention* mechanism affects these objectives.

To evaluate the extent of performance enhancement, we removed the *Attention* layer (Section 5.4) of *Learning Engine* and repeated the experiment in Section 7.2.1. The result of this experiment is shown in Table 2 under the *ACETON without Attention* column. As corroborated by these results, **removing the *Attention* negatively impacts the precision and recall values**. For example in *Network* category, the recall drops to 88.75% compared to 100% in *ACETON with Attention*, i.e., the model misses 9 out of 71 Network defects. Removing *Attention* from ACETON also negatively impacts training time. That is, it takes longer for the model to learn the patterns and converge. We discuss this more in RQ5.

Attention Analysis produces a set of features as output on which the oracle has attended more. To visually confirm that ACETON has attended to relevant features for each category of energy defects, i.e., to determine its validity, we created the heatmap shown in Figure 6. The horizontal axis of heatmap indicates SV, while the vertical axis indicates subcategories listed in Table 1. To construct the heatmap, we counted the appearance of each attended feature for all its instances in a subcategory, and divided it by the occurrence of all the attended features under that subcategory to define a weight for it. The weights take a value between $(0, 1]$ and the higher is the weight for a feature, the model attended to it more under the given subcategory, thus its corresponding color in heatmap is closer to yellow.

As the heatmap clearly shows, the hot areas of heatmap for each subcategory in the vertical axis maps to its corresponding category in the SV, meaning that the model has attended to relevant features to decide the output of tests. An interesting observation from this heatmap is that lifecycle features, specifically *Activity Paused*, *Activity Destroyed*, and *Service Stopped*, frequently appear in the attended features. This shows that energy defects are not solely related to the changes in app or hardware states, but a combination of both.

Finally, we aggregated the list of attended features for each category and formally specified them, as shown in Table 1 under *Defect Signature* column. While our intention for deriving defect signatures was to verify the validity of the DL model, we believe that the ability of ACETON to extract and formalize the signatures can further help developers to localize the energy defects, specifically for new types of energy defects that will emerge as Android framework evolves. For example, the signature of *Unnecessary Active Bluetooth Connections* shows the root cause of this issue is failing to close a Bluetooth connection ($BC = 1$) when the Bluetooth is off or turning off ($BE = 0$), which causes battery consumption even when the app is paused ($AP = 1$) or destroyed ($AD = 1, SS = 1$).

7.4 RQ3: Detecting Unseen Defect Types

While prior research question evaluated effectiveness of ACETON in detection of defect types it was trained on, this research question investigates its ability to detect previously *unseen* defect types. Generally speaking, DL models can only predict patterns that they have been trained on. However, we hypothesize that if our oracle is trained on a subset of defect types *for a specific hardware element*, it may be able to detect unseen defect types *related to that hardware* as well. To that end, we excluded one subcategory listed in Table 1 at a time, trained the model on the energy defects related to all other subcategories among all hardware categories, and used instances of the excluded subcategory as test data.

Here, we use recall as an evaluation metric to evaluate effectiveness of ACETON. Precision is not a proper metric here, since our test data only belongs to one subcategory (class) in this experiment and no false positive is generated. Column *Unseen Recall* in Table 1 shows the result for this experiment. We can see that in the majority of the cases **ACETON is able to effectively detect previously unseen energy defect types**. In fact, the recall value for majority of the excluded sub-categories is above 93%. However, there are a few subcategories with lower recall values, which are marked by * in Table 1. These are the cases in which the attended features, i.e., defect signature, is drastically different from that of in the training dataset. We believe as additional energy defects are included in the training dataset of ACETON, its ability to detect previously unseen energy defects can improve too.

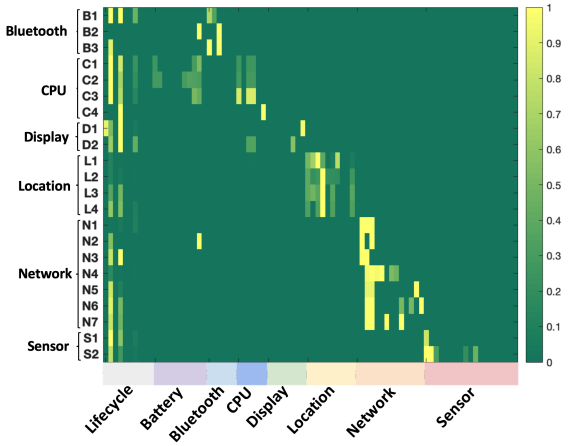

Figure 6: A heatmap representing the attended features of SV for different subcategories of energy defects

7.5 RQ4: Reusability of the Oracle

In answering prior research questions, we showed that the oracle generated by ACETON is reusable among different apps. Here, we investigate if the oracle is also reusable across different mobile devices. Experiments in prior research questions were performed on a Google Nexus 5X phone, running Android version 7.0 (API level 24). For this experiment, we used an additional phone, Nexus 6P, running Android version 6.0.1 (API level 23). These two devices are not only different in terms of Android version, but they also have different hardware configurations, e.g., different pixel density and resolution for Display, CPU frequency, RAM size, Battery capacity, etc.

We first repeated the experiments in Section 7.2.1 on the new device to ensure that the oracle model is still effective in detecting energy defects. The result of this experiment showed the same level of precision and recall for the new oracle (average precision = 98.27%, average recall = 99.48%). Afterwards, we wanted to see if the oracle trained on one device can correctly predict the label of tests executed on the other device.

To that end, we split the instances of *Labeled Database* into two subsets, 90% of them to be used for training and the remaining 10% for testing. Next, we trained two oracles on the mentioned devices, $oracle_1$ on Nexus 5x device and $oracle_2$ on the Nexus 6P device, by executing the instances in the training set and collecting their sampled SVs on the corresponding device. Similarly, we executed instances of test dataset on both devices, $test_1$ on Nexus 5x and $test_2$ on the Nexus 6P. We then evaluated $test_1$ using $oracle_2$ and $test_2$ using $oracle_1$. The average precision and recall values for $test_1$ on $oracle_2$ are 99.95% and 99.81%, respectively. Similarly, $oracle_1$ was able to detect the labels for $test_2$ with an average precision of 99.89% and recall of 99.45%. These results confirm that our energy oracles are device independent, hence, reusable.

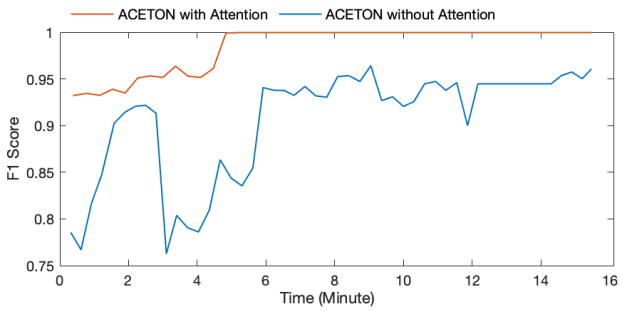

Figure 7: F1 Score of ACETON with and without Attention captured during the training phase

7.6 RQ5: Performance

To answer this research question, we evaluated the time required to train and test the oracle. We ran the experiments on a laptop with 2.2 GHz Intel Core i7 CPU and 16 GB RAM. It took 4.5 minutes on average for ACETON to train an energy oracle on the whole dataset, while it took only 37.6 milliseconds on average for the trained oracle to predict the label of tests in our experiments. In addition, we examined to what extent *Attention Mechanism* speeds up ACETON's learning. To that end, we disabled the early-stopping criterion (recall Section 7.1) and tracked the *F1 Score* of the following two models during their training: *ACETON with Attention* and *ACETON without Attention*. As shown in Figure 7, *ACETON without Attention* requires more time to train a model that achieves a comparable F1 Score as *ACETON with Attention*. In fact, even after 14 minutes of training, *ACETON without Attention* was not able to match the F1 Score of *ACETON with Attention*. These results confirm that ACETON is sufficiently efficient for practical use.

8 RELATED WORK

Automated test oracle approaches in the literature can be categorized into *Specified* [16, 17, 20–23, 35, 38, 41, 44], *Derived* [24, 45, 46], and *Implicit* [12, 14, 15, 25, 36, 36, 42] test oracles [11]. Majority of these technique focus on the functional properties of the program to generate test oracles, e.g., generating test oracles for GUI. Even among those that consider non-functional properties of software [12, 15, 25, 36, 42], none has aimed to develop an oracle for energy testing. ACETON is the first attempt to construct automated, reusable energy test oracles for mobile apps. To the best of our knowledge, it is also the first effort of using Deep Learning to tackle the oracle problem.

The biggest challenge to construction of an energy oracle is determining the observable patterns during test execution that are indicators of energy defects. While prior research attempted to categorize energy defects in mobile apps, the proposed fault models are either broadly describing a category of energy defects [10], or identifying specific energy anti-patterns in code that lead to excessive battery consumption [27, 29–31]. Also, as energy defects change and new types of defects emerge due to the evolution of mobile platform, i.e. Android framework, the defect model proposed by prior work becomes obsolete. ACETON's contribution is the ability to automatically learn the changes in the state of hardware and environmental settings with high precision and recall, even for unseen patterns.

The closest approaches to ACETON in terms of detecting energy defects through testing are Jabbarvand et al. [28, 29], and Banerjee et al. [9, 10]. μDroid [29] is an energy-aware mutation testing framework for Android. It implements 50 energy mutants and relies on comparing power traces of original and mutant versions of an app to construct an oracle, i.e., to determine whether a test kills a mutant or not. The proposed technique for construction of mutation testing oracle in [29] cannot be generalized to energy test oracles, as it requires a baseline power trace—that of original app—to identify anomalous patterns in a given power trace—mutant app.

Cobweb [28] is a search-based energy testing framework for Android. The proposed approach employs a set of models to take execution context into account, i.e., lifecycle and hardware state context, in the generation of tests that can effectively find energy defects. While Cobweb is effective for generating energy-aware test inputs, it does not address the automatic construction of oracles for energy tests.

Banerjee et al. [10] presents a search-based profiling strategy with the goal of identifying energy defects in an app. They construct a graph representing an app's GUI events, extract the event traces using the generated graph, and explore event traces that may possibly reach energy hotspots, while profiling energy consumption of the device. In fact, [10] analyzes the power traces using statistical and anomaly detection techniques to uncover energy-inefficient behavior. Unlike the automated oracles generated by ACETON, usage of a power measurement hardware makes their approach device dependent, expensive, and thereby impractical.

In their subsequent work [9], Banerjee et al. fixed the scalability issue of the prior work [10] by using abstract interpretation-based program analysis to detect resource leaks. Similar to the prior work, they rely on a dynamically constructed model for GUI events to guide the search for finding paths leading to a resource leak. Unlike ACETON's test oracles that are reusable and can detect a wide range of energy defects, test oracles generated by EnergyPatch are specifically targeted to detection of resource leaks.

9 CONCLUDING REMARKS

Energy efficiency is an increasingly important quality attribute for mobile apps that should be properly tested. Recent advancements in energy testing have in large part focused on test input generation, and not on the automated construction of test oracles. The key challenge for the construction of energy test oracles is derivation of reusable patterns that are indicative of energy defects. We presented ACETON, the first approach for automated construction of energy test oracles that leverages Deep Learning techniques to learn such patterns. Our experimental results show that the energy oracle constructed using ACETON is highly reusable across mobile apps and devices, achieves an overall accuracy of 99%, and efficiently detects the existence of energy defects in only 37 milliseconds on average.

10 ACKNOWLEDGEMENT

This work was supported in part by awards 1823262 and 1618132 from the National Science Foundation and a Google PhD Fellowship.

REFERENCES

[1] 2019. Android Broadcasts Overview. https://developer.android.com/guide/components/broadcasts

[2] 2019. Android Service Overview. https://developer.android.com/guide/components/services

[3] 2019. Location Manager Strategies. https://developer.android.com/guide/topics/location/strategies.html

[4] 2019. Monsoon Power Monitor. https://www.msoon.com/

[5] 2019. Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler

[6] 2019. Understanding Android Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle

[7] 2020. ACETON tool and artifacts. https://seal.ics.uci.edu/projects/aceton/index.html

[8] Dzmitry Bahdanau et al. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[9] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2018. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering* 44, 5 (2018), 470–490.

[10] 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.

[11] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2015), 507–525.

[12] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2011. Finding software vulnerabilities by smart fuzzing. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 427–430.

[13] Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166.

[14] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 10.

[15] Mariano Ceccato, Cu D Nguyen, Dennis Appelt, and Lionel C Briand. 2016. SOFIA: An automated security oracle for black-box testing of SQL-injection vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 167–177.

[16] Yoonsik Cheon. 2007. Abstraction in assertion-based test oracles. In *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE, 410–414.

[17] Yoonsik Cheon and Gary T Leavens. 2002. A simple and practical approach to unit testing: The JML and JUnit way. In *European Conference on Object-Oriented Programming*. Springer, 231–255.

[18] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[19] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2019. Greenscaler: training software energy models with automatic test generation. *Empirical Software Engineering* 24, 4 (2019), 1649–1692.

[20] David Coppit and Jennifer M Haddox-Schatz. 2005. On the use of specification-based assertions as test oracles. In *29th Annual IEEE/NASA Software Engineering Workshop*. IEEE, 305–314.

[21] Marie-Claude Gaudel. 2001. Testing from formal specifications, a generic approach. In *International Conference on Reliable Software Technologies*. Springer, 35–48.

[22] Gregory Gay, Sanjai Rayadurgam, and Mats PE Heimdahl. 2014. Improving the accuracy of oracle verdicts through automated model steering. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 527–538.

[23] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. [n.d.]. Automatic Generation of Oracles for Exceptional Behaviors. ([n. d.]).

[24] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. 2000. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing Verification and Reliability* 10, 3 (2000), 171–194.

[25] David L Heine and Monica S Lam. 2003. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 168–181.

[26] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[27] Reyhaneh Jabbarvand. 2017. Advancing energy testing of mobile applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 491–492.

[28] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. 2019. Search-based energy testing of Android. In *ICSE 2019*. IEEE Press, 1119–1130.

[29] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: an energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 208–219.

[30] 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.

[31] Reyhaneh Jabbarvand Behrouz. 2020. *Advancing Energy Testing of Mobile Applications*. Ph.D. Dissertation. UC Irvine.

[32] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 15.

[33] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[34] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*.

[35] Ying-Dar Lin, Jose F Rojas, Edward T-H Chu, and Yuan-Cheng Lai. 2014. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering* 40, 10 (2014), 957–970.

[36] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. 2013. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1012–1021.

[37] 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.

[38] Atif M Memon, Martha E Pollack, and Mary Lou Soffa. 2000. Automated test oracles for GUIs. In *ACM SIGSOFT Software Engineering Notes*, Vol. 25. ACM, 30–39.

[39] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[40] Lutz Prechelt. 1998. Early stopping-but when? In *Neural Networks: Tricks of the trade*. Springer, 55–69.

[41] Chunhui Wang, Fabrizio Pastore, and Lionel Briand. 2018. Oracles for Testing Software Timeliness with Uncertainty. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2018), 1.

[42] Elaine J Weyuker and Filippos I Vokolos. 2000. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering* 26, 12 (2000), 1147.

[43] Claas Wilke, Sebastian Richly, Sebastian Gotz, Christian Piechnick, and Uwe Aßmann. [n.d.]. Energy Consumption and Efficiency in Mobile Applications: A user Feedback Study. In *The Internation Conf. on Green Computing and Communications*.

[44] Qing Xie and Atif M Memon. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16, 1 (2007), 4.

[45] Tao Xie. 2006. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*. Springer, 380–403.

[46] Tao Xie and David Notkin. 2005. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering* 31, 10 (2005), 869–883.

[47] Zhilu Zhang and Mert Sabuncu. 2018. Generalized cross entropy loss for training deep neural networks with noisy labels. In *Advances in neural information processing systems*. 8778–8788.