Search-Based Energy Testing of Android

Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
{jabbarvr,junwel1,malek}@uci.edu

Abstract—**The utility of a smartphone is limited by its battery capacity and the ability of its hardware and software to efficiently use the device's battery. To properly characterize the energy consumption of an app and identify energy defects, it is critical that apps are properly tested, i.e., analyzed dynamically to assess the app's energy properties. However, currently there is a lack of testing tools for evaluating the energy properties of apps. We present COBWEB, a search-based energy testing technique for Android. By leveraging a set of novel models, representing both the functional behavior of an app as well as the contextual conditions affecting the app's energy behavior, COBWEB generates a test suite that can effectively find energy defects. Our experimental results using real-world apps demonstrate not only its ability to effectively and efficiently test energy behavior of apps, but also its superiority over prior techniques by finding a wider and more diverse set of energy defects.**

I. INTRODUCTION

Improper usage of energy consuming hardware elements, such as GPS, WiFi, radio, Bluetooth, and display, can drastically discharge the battery of a mobile device. Recent studies have shown energy to be a major concern for both users [1] and developers [2]. In spite of that, many mobile apps are abound with energy defects. This can be attributed to the lack of tools and methodologies for energy testing [2]. Recent advancements in mobile app testing have mostly focused on testing functional correctness of programs, which may not be suitable for revealing energy defects [3]. There is, thus, an increasing demand for solutions to assist developers in testing energy behavior of apps prior to their release.

The first step toward energy testing is to understand the properties of tests that are effective in revealing energy defects in order to automatically generate such tests. Recently, Jabbarvand et al. [3] proposed a technique based on mutation testing to identify the properties of proper tests for energy testing. They showed that to kill the energy mutants, tests need to be executed under a variety of *contextual* settings. Based on the results of their study, we have identified three contextual factors that are correlated to energy defects and should be considered in energy-driven testing: **(1) Lifecycle Context**: A subset of energy defects, e.g., wakelocks and resource leaks, manifest themselves under specific sequences of lifecycle callbacks; **(2) Hardware State Context**: Some energy defects happen under peculiar hardware states, e.g., poor network signal, no network connection, or low battery; and **(3) Interacting Environment Context**: Certain energy defects manifest themselves under specific interactions with the environment—consisting of user, backend server, other apps, and connected devices such as smartwatches.

None of the prior automated Android testing techniques properly consider these contextual factors in test generation [4], [3], thereby are not able to effectively test the energy behavior of apps. That is, majority of the state-of-the-art Android testing tools [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] are aimed for GUI testing, which only considers the inputs directly generated by user, e.g., clicking on a button. Even among the techniques that go beyond GUI testing [16], [17], there is no systematic approach for altering the lifecycle of components and state of hardware elements to properly evaluate the energy behavior of apps.

In this paper, we present COBWEB, an energy testing technique for Android apps. COBWEB uses an evolutionary search strategy with an energy-aware genetic makeup for test generation. By leveraging a set of novel models, representing lifecycle of components and states of hardware elements on the phone, COBWEB is able to generate tests that execute the energy-greedy parts of the code under a variety of contextual conditions. Extensive evaluation of COBWEB using real-world Android apps with confirmed energy defects demonstrates not only its ability to effectively and efficiently test energy behavior of apps, but also its superiority over prior techniques by finding a wider and more diverse set of energy defects.

The remainder of this paper is organized as follows. Section II introduces an illustrative example that is used to describe our research. Section III provides an overview of our approach, while Sections IV-V describe the details. Section VI presents the evaluation results. The paper concludes with a discussion of the related research and avenues of future work.

II. ILLUSTRATIVE EXAMPLE

As an illustrative example, we use an Android app called *MyTracker* [18]. In this section, we describe two main functionalities of MyTracker, two tests to exercise these functionalities, and two energy defects in this app that cannot be caught by tests that do not take execution context into account.

App: As shown in Figure 1, MyTracker allows users to search for the map of different locations using either the internet or GPS, download them, and navigate through each specific downloaded map. This app consists of seven components, i.e., four *Activities* and three *Services*. MyTracker provides two functionalities: tracking/navigation and search/download map.

When a user clicks on the *Download Maps* button, the app navigates to `MapSearchActivity`, where the user can search for maps using the Internet or GPS. If the user decides to search using the Internet, she needs to provide the name of the city, e.g., Ottawa, and then click on the `Find by Internet` button. Otherwise, she can just click on the `Find by GPS` button. Depending on the selected search option, the app starts `InternetService` or `GPSService` in the background, which searches for the map on a specific server. Upon finding a

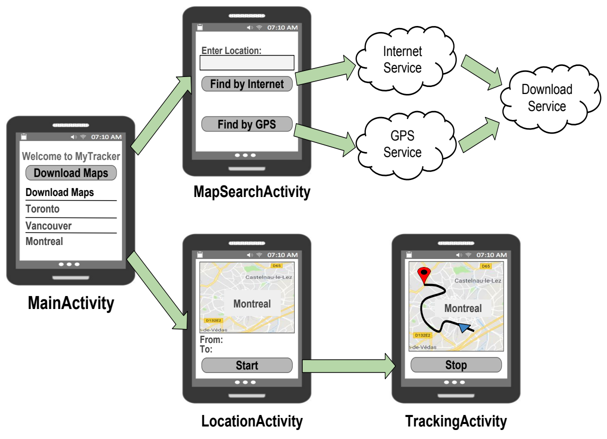

Fig. 1: MyTracker Android Application

match with the name provided by user or location coordinates, `DownloadService` downloads the map, resulting in the list of maps displayed on `MainActivity` to be updated.

For tracking, once the user clicks on one of the downloaded maps in `MainActivity`, e.g. the map of *Montreal* shown in Figure 1, the app navigates to `LocationActivity`. In this activity, the user can see the map of *Montreal* and provide a source and destination address to start the navigation. By clicking on the `Start` button, the app starts `TrackingActivity` and registers a location listener, which updates the GUI of `TrackingActivity` upon movement.

Tests: Android tests can be represented as a sequence of events, where each event is an input to the app and can be triggered by the user or system. We formally define each test t in test suite T, as $\langle c_1\langle e_1,\dots,e_{p_1}\rangle,\dots,c_m\langle e_1,\dots,e_{p_m}\rangle\rangle$, where c_i indicates the *ith* component covered during the execution of t. The execution of each component c_i, which could be Activity, Service, or Broadcast Receiver, by test t is represented as an event sequence, where each event is denoted as e. We consider two types of events: (1) input events that take inputs using specific APIs, e.g., filling a text box, and (2) callback events that are invocation of Android callbacks, e.g., click on a button or transition to a lifecycle state. Figure 2 shows representation of two tests according to this formalism that target the two functionalities of MyTracker app. We use these tests throughout the paper for illustrating our approach.

Energy Defects: MyTracker suffers from two energy defects: 1) *Fail to check connectivity* energy defect [3] occurs when an app fails to check for connectivity before performing a network operation. MyTracker unnecessarily searches for a network signal when there is no network connection available, which is a power draining operation. To find this energy defect, MyTracker should be tested both when there is a network connection available and not. The test corresponding to Sequence 1 in Figure 2 does not enable or disable network connectivity, therefore, cannot detect this defect.

Sequence 1: <**MainActivity**<onCreate, onClick("Download Maps")>, **MapSearchActivity**<onCreate, enterText("Ottawa"), onClick("Find by Internet")>, **InternetService**<onStartCommand, searchOnServer, startService("DownloadService")>, **DownloadService**<onStartCommand, startDownload, onDownloadComplete>>

Sequence 2: <**MainActivity**<onCreate, onItemClick("Montreal")>, **LocationActivity**<onCreate, enterText("airport"), enterText("conference"), onClick("Start")>, **TrackingActivity**<onCreate, onLocationChanged("location1"), onClick("Stop")>>

Fig. 2: Event sequences for testing the tracking/navigation and search/download functionalities of MyTracker

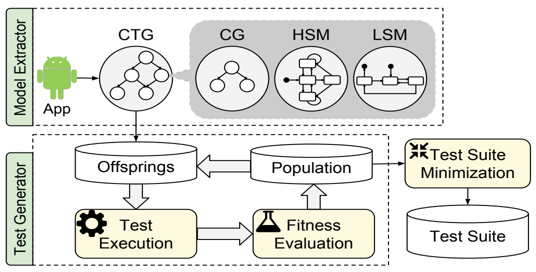

Fig. 3: COBWEB Framework

2) MyTracker starts listening to location updates in `TrackingActivity` by registering a location listener for GPS. As long as `TrackingActivity` is visible to the user and GUI is rendered based on location updates, MyTracker can keep the GPS active. However, when user puts the app in the *Paused* state, i.e., MyTracker is sent to background, it does not unregister the location listener, thereby, unnecessarily updates a GUI that is not visible to the user [19], [20]. To find this energy defect, a test needs to put `TrackingActivity` into paused state for some time to assess utilization of GPS hardware in this state. Clearly, the test corresponding to Sequence 2 in Figure 2 does not have this property.

III. APPROACH OVERVIEW AND CHALLENGES

Since the domain of events and inputs for android apps is quite large, COBWEB follows a search-based testing technique for input generation. Every search-based testing technique has three facets: (1) *search space*, which is a set of possible solutions, (2) *meta-heuristics* to guide the search through the search space, and (3) *evaluation metrics* to measure the quality of potential solutions.

COBWEB identifies the search space as a set of event sequences, i.e., system tests. To guide the search through the search space, our approach utilizes an evolutionary algorithm to globally search for an optimal solution. Similar to other search-based techniques, COBWEB relies on the abstract representation of the program, i.e., models, to generate event sequences and compute the fitness function as an evaluation metric. However, a key novelty of COBWEB is that unlike prior search-based testing techniques, it also utilizes several other contextual models, representing the state of hardware and environment, in the search process.

Figure 3 provides an overview of COBWEB, consisting of two major components: (1) *Model Extractor* component that derives the required models for test generation; and (2) *Test Generator* component that utilizes an evolutionary search-based technique to create system tests. COBWEB's fitness function rewards the tests based on two criteria: (1) how close they are to covering energy-greedy APIs in the application logic, and (2) how well they contribute to exercising different contextual factors. There are three main challenges that COBWEB should overcome:

Invalid or useless tests: The order of events is important for exercising specific behaviors in apps. For example, a common approach to test whether an app like MyTracker is properly utilizing GPS is to mock location/movement. However, mocking can only produce a callback on the app if the app under test has already registered a location listener. Otherwise, mocking the location is useless, as it cannot test the usage of GPS by the app. In addition, prior research has shown genetic operations, such as cross over, may produce many *invalid* event sequences that fail to execute [10]. To reduce the generation of invalid or useless tests, COBWEB relies on two models representing the app's functional behavior, namely *Component Transition Graph (CTG)* and Call Graph (CG).

Contexual factors: In addition to the models that represent the app's functional behavior, further models are required to take the execution context into account during test generation. COBWEB uses two additional models, namely *Lifecycle State Machine (LSM)* and *Hardware State Machine (HSM)* to account for contextual factors.

Scalability: Search-based techniques are susceptible to generation of large number of tests [21], [22], which can pose a scalability barrier due to the time consuming fitness evaluation. Although fitness evaluation can be performed in parallel [23], [24], it entails usage of distributed devices or special multi-core PCs. The majority of mobile apps are developed at a nominal cost by entrepreneurs that do not have such resources. To tackle the scalability issue during test generation, COBWEB generates intermediate tests in the form of *Robolectirc* tests [25], which can be executed atop JVM very fast. The final test suite is transformed to *Espresso* [26] tests that can be executed on emulator or mobile devices.

IV. MODEL EXTRACTOR

COBWEB uses four types of models: *Component Transition Graph (CTG)*, *Call Graph (CG)*, *Lifecycle State Machine (LSM)*, and *Hardware State Machine (HSM)*. Figure 4 shows a subset of these models for MyTracker app. At the highest-level is the CTG model, which represents the components comprising the app as nodes and the *Intents* as transitions among the nodes. Intents are Android events (messages) that result in the execution flow to move from one component to a different component. Each node of the CTG in turn contains one CG—representing the internal behavior of the corresponding software component, one LSM—representing the possible lifecycle states of the corresponding software component, and zero or more HSM—each of which represents the states of an energy-greedy hardware element utilized during the execution of the corresponding software component. LSM and HSM models are generic and app/device independent, constructed manually by the authors, while CTG and CG models are app-specific and automatically extracted through static analysis of an app's bytecode. We describe each model and how it is obtained in the remainder of this section.

A. Component Transition Graph (CTG)

COBWEB utilizes CTG to ensure generation of valid and useful event sequences. Events can be categorized into (1) *input events* that take inputs to the app using specific APIs, e.g., `EditText.getText()` that reads a string provided by user for a text box, and (2) *callback events* that invoke Android callbacks, e.g., `onLocationChanged()`, which is invoked when the physical location of the device changes.

COBWEB uses CTG model of the app under test to generate the proper order of event calls. Finding the proper order of event call invocations is particularly a challenge in Android due to usage of callbacks, each considered a possible entry point for an application. For example, `onLocationChanged()` callback is an entry point for MyTracker app. The call graph obtained from running the state-of-the-art static analysis tools, such as Soot [27], does not model any particular order for the execution of entry points. That is, using such call graphs to generate event sequences, `onLocationChanged()` can appear before the `onCreate()` of `TrackingActivity` or even `onCreate()` of `MainActivity`. However, proper invocation of `onLocationChanged()` is after the execution of `onCreate()` of `TrackingActivity`, as shown in Sequence 2 of Figure 2.

Furthermore, to properly test the energy behavior of MyTracker with respect to its tracking functionality, COBWEB needs to mock the location, such that Android platform invokes `onLocationChanged()` callback. The tricky part of generating such tests is that `onLocationChanged()` callback should only be invoked if the app has already registered a location listener to receive location updates, which happens in the `onCreate()` method of `TrackingActivity` component. In other words, mocking the location should be performed after `TrackingActivity` starts. Otherwise, mocking has no effect and will not result in the invocation of `onLocationChanged()` callback. Generating valid and useful events entails not only an inter-procedural analysis—to find the proper component for callback invocation—but also requires considering the specific types of dependencies among events. To overcome these challenges, CTG considers *five* types of transitions:

1- **Call transition**: These intra-component transitions are inferred from the basic call graph generated for the app under test using Soot [27].

2- **Intent transition**: These transitions are inter-component, which result in transferring the control from one component to another component. A method or callback inside one component that starts another component is connected to the lifecycle entry point of that component using this kind of transition. COBWEB uses IC3 [28] to infer Intent transitions.

3- **GUI transition**: These intra-component transitions indicate the order of execution between GUI widgets. For example, the *Start* button in the `LocationActivity` of MyTracker should be clicked after user provides source and destination addresses in the *From* and *To* text boxes. COBWEB builds on top of TrimDroid [11] to infer such transitions.

4- **Registration transition**: This type of transition consists of two sub-categories: *broadcast receiver registration* and *event listener registration*. A broadcast receiver receives an intent for which it has registered for via the `onReceive()` callback method. While static broadcast receivers—those identified in the manifest file—are registered when the app launches, dynamic broadcast receivers are registered using `registerReceiver()` API. Broadcast registration transition, which could be inter- or intra-component, connects a CG node that registers a broadcast receiver to its corresponding `onReceive()` callback, which is also a CG node.

An event listener is an interface that contains one or more callbacks. Listener callbacks are called by the Android framework when the event that the listener has been registered for is triggered either by user or environment. For example, `onLocationChanged()` is called upon any change in the location of the device, if the app has previously registered a location listener. Listener registration transition, which could also be inter- or intra-component, connects a CG node that registers a listener to its corresponding callbacks, which is also a CG node. The listener callbacks have no order among themselves.

COBWEB's approach for identifying registration transition works as follows. For a given registered callback, COBWEB performs an inter-procedural, flow-sensitive static program analysis to find the *registrar*—the entity that registers the broadcast receiver or listener of that callback. It then assigns a transition from the registrar to the registered callback node in CG. For broadcast registration, the registered callback is `onReceive()`—either defined inside an inner-class broadcast receiver or a broadcast receiver component, and registrar is callback or method that invokes the `registerReceiver()` API. For listener transition, COBWEB takes a list of listener callbacks available in Android API[1] to identify registered callbacks. The listener registrar is a callback or method that registers a listener with the given callback implemented. Flow-sensitivity is required for this analysis, as a broadcast receiver may subscribe to receive multiple Intents, and multiple listeners of the same kind might be registered for an app. For example, an app may register two location listeners, one listening to GPS location updates, and another one tracking location updates via network.

5- **Lifecycle transition**: These intra-component transitions are between starting lifecycle callback nodes of a component, e.g., `onCreate()` for Activities or `onStartCommand()` for Services, and every non-lifecycle node with no incoming edge inside the component. That is, every callback or method inside a component with no incoming edge can be called after the component is started. COBWEB resolves lifecycle transitions after all other transitions are identified. It ignores all other lifecycle callbacks that do not instantiate/start a component, e.g., `onPause()` or `onDestroy()`, since these other lifecycle callbacks are considered using the LSM model, discussed next.

B. Lifecycle State Machine (LSM)

Wakelocks and other resources, such as GPS, are commonly acquired and released in lifecycle event handlers [29]. Thereby, proper implementation of lifecycle callbacks is important, as developers need to ensure apps are not unnecessarily consuming power due to changes in the lifecycle state. To that end, we represent possible transitions among lifecycle states of an Android component type as a finite state machine, called Lifecycle State Machine (LSM).

Since the lifecycle callbacks are handled by the Android framework itself, we can define an LSM for each Android component type, regardless of which callbacks are actually implemented by instances of that component. Such a representation also ensures thorough testing of an app, as developers may have failed to implement important lifecycle callbacks, where resources should be managed properly.

We derived three types of LSM models, one for each of the Android components types (Activities, Services, and Broadcast

[1]Derivation of this list is discussed in Section IV-C

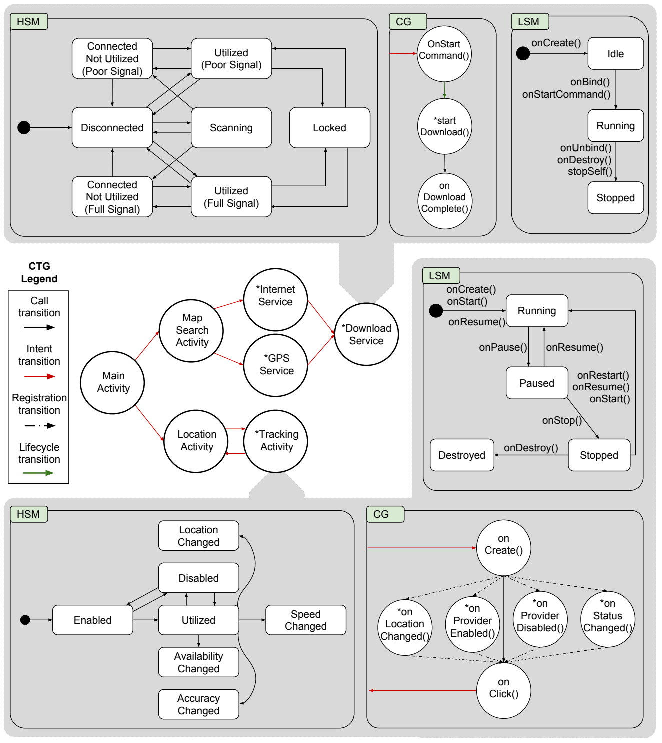

Fig. 4: CTG model for MyTracker. Gray boxes show the detailed CG, LSM, and HSM of *DownloadService* and *TrackingActivity* components. Components marked with an asterisk contain energy-greedy API invocations

Receivers), based on the lifecycle callbacks identified for them in the Android documentation. Figure 4 shows LSMs of the Activity and Service components for `TrackingActivity` and `InternetService`, respectively. For example, the Activity LSM demonstrates four different lifecycle states for an Activity component. The Activity LSM indicates how the execution of lifecycle callbacks results in transitions to different states.

C. Hardware State Machine (HSM)

Developers should adjust the functionality of apps according to the states of hardware elements. For instance, per Android developer guidelines [19], a location listener should be unregistered when user is stationary, or the frequency of location update should be lowered when user is walking rather than driving. To take such factors into account, we need to look for changes in the hardware states from the inputs generated by the environment (e.g., change in the strength of network signal), or the user, directly or indirectly (e.g., user can turn on/off location directly from setting, or she can trigger changes in the state of GPS by changing her location).

Identifying different states of hardware elements for energy testing is crucial, since apps consume different amounts of energy in different states [30]. We followed a systematic approach to derive generic and reusable models for each hardware element on a mobile device, called Hardware State Machine (HSM).

Android provides libraries to access and utilize hardware elements. These libraries provide APIs and constant values, i.e., fields, which can be used to inquire about possible states of hardware elements. Developers can use the APIs implemented by such libraries to monitor the state of hardware elements (e.g., using `LocationManager` to track user location changes and `ConnectivityManager` to query about the state of network connections) or manipulate the states (e.g., hold a lock on the CPU using `PowerManager.Wakelock` APIs to prevent the phone from going to sleep). Documentation of these APIs is a rich source for identifying different hardware states.

Similarly, constant values introduced in such libraries can be used to identify hardware states, as they usually are either representative of different states of hardware elements, or the *action* field of broadcast Intents that show a change in the state of hardware. For example, `WIFI_MODE_FULL`, `WIFI_MODE_FULL_HIGH_PERF`, and `WIFI_MODE_SCAN_ONLY` are constants associated with `WiFiManager` library, indicating that WiFi hardware can operate in different modes, each consuming battery of the device differently.

To find a thorough list of such libraries, we started by automatically crawling Android API reference [31] using Crawler4J [32] to search for classes, where description of their public methods or fields contained at least two of the following keywords: location, lock, gps, network, connect, radio, cellular, bluetooth, display, sensor, cpu, battery, power, consume, drain, charge, discharge, monitor, hardware, state, and telephone. We crawled $6,279$ pages in total and collected $1,971$ libraries after keyword filtering. We further processed the documentation of those libraries to find all the possible states of hardware elements as follows:

1. APIs: To automatically collect a set of APIs that *monitor* state of the hardware elements, we searched for event listeners and callbacks in the public methods of the $1,971$ collected libraries, as they monitor the changes in the state of hardware elements. From a total of $38,626$ APIs in these classes, we searched for APIs that have the keyword *listener* in their signature—for event listener APIs—and APIs that start with *on*—for callbacks. This yielded 441 listeners and $2,968$ callbacks. To collect APIs that *manipulate* state of hardware, we searched for methods that have derivation of the following keywords in their description: scan, access, acquire, release, state, register, disable, enable, connect, and disconnect. In the end, we collected a total of 104 APIs correlated to different states of various hardware elements.

2. Fields: We automatically searched for the constant values identified for the collected libraries, whose description contained one of the keywords we used for initial filtering. This search left us with 225 constant values.

Once the states of each hardware element were identified using the aforementioned approach, we constructed seven HSMs for major hardware elements on mobile phones. These HSMs correspond to battery, Bluetooth, CPU, display, GPS, radio, and sensors, e.g., accelerometer and gyroscope.

HSM is a finite state machine that represents different states of a hardware element. Figure 4 shows HSM models derived for Network and Location hardware elements (in the details of `InternetService` and `TrackingActivity` components). For Network HSM for example, from 46 APIs and 12 fields of two libraries—`ConnectivityManager` and `WiFiManager`—along with their nested classes, we identified

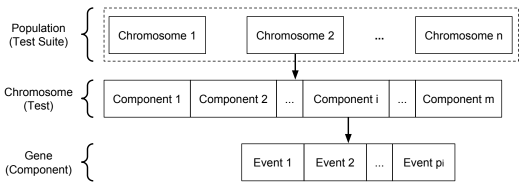

Fig. 5: Genetic representation of tests

Algorithm 1: Evolutionary Energy Test Generation

```
Input: App $app$, Set of $LSM$s, Set of $HSM$s, List of
       energy-greedy APIs $HW, threshold, breedSize$
Output: Test suite $T_E$
1  $CTG, CG \leftarrow staticAnalysis(app)$
2  $model \leftarrow mergeModels(CTG, CG, HSM, LSM)$
3  $P \leftarrow randomPopulation(app)$
4  while improvement in fitness$(T_R, model) \leq threshold$ do
5      $P_{offspring} \leftarrow select(P, breedSize)$
6      $P_{offspring} \leftarrow converge(model, P_{offspring})$
7      $P_{offspring} \leftarrow diverge(model, HW, P_{offspring})$
8      $T_{R_{tmp}} \leftarrow generate(P_{offspring})$
9      $fitness(T_{R_{tmp}}, model)$
10     $P \leftarrow merge(P, P_{offspring})$
11     $T_R \leftarrow T_R \cup T_{R_{tmp}}$
12 $T_E \leftarrow minimize(T_R)$
```

9 states for Network, namely *Disconnected*, *Connected* (with poor or full signal strength), *Utilized* (under poor or full signal strength), *Scanning*, and *Locked* (full, multi-cast, and high performance).[2] Edges between different states of the hardware can be traversed by calling one of the Android APIs inside the app or triggering events outside of it.[3] Hence, it is crucial to have a generic HSM for each hardware without considering just the source code of the app. For example, an application can start scanning for available WiFi networks using `startScan()` API, or the state of hardware can be changed to scanning by manipulating the platform. We have made the HSM models of other hardware elements publicly available [33].

V. Test Generator

Our objective is to generate tests that (1) cover energy-greedy APIs, and (2) execute them under different contextual conditions. In this section, we describe the evolutionary search-based test generation algorithm utilized in COBWEB that aims to satisfy this objective.

A. Genetic Algorithm

COBWEB identifies the search space for energy testing problem as a set of system tests. Figure 5 illustrates the genetic representation of a test suite generated by COBWEB. Overall, COBWEB generates a set of system tests that corresponds to a population of *chromosomes* in the evolutionary algorithm. At a finer granularity, each chromosome consists of *genes*, which are the main Android components of an app, and each gene contains multiple *sub-genes*, which are either input events or callback events (recall Section II).

Algorithm 1 presents the evolutionary approach of COBWEB for test generation. It takes the app along with LSM and

[2]For a better illustration, different locked states are merged in the HSM
[3]Labels of edges are not shown here for sake of simplicity

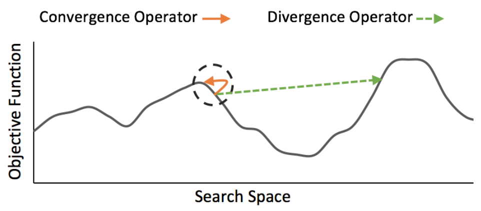

Fig. 6: Intuition behind convergence and divergence operators

HSM models as inputs and generates a set of Espresso [26] tests—T_E. The algorithm starts by constructing the CTG and CG models through static analysis of the app (Line 1) and integrating those with LSM and HSM models to arrive at the final $model$ of the app under test (Line 2). Next, it randomly generates the initial population P, which is later evolved using evolutionary search operators through multiple iterations (Lines 5-7). Once the new generation is available, COBWEB generates Robolectric tests for each chromosome (Line 8), executes them on JVM, and calculates their corresponding fitness (Line 9). At the end of iteration, COBWEB adds newly generated tests to the test suite and starts a new iteration. This process continues until the termination condition is met: if the improvement in the average fitness of generated tests in two consecutive iterations is less than a configurable $threshold$, the algorithm terminates (Line 4). Afterwards, Algorithm 1 minimizes the generated Robolectric test suite and transforms them to Espresso tests for execution on a mobile device (Line 12), such that energy measurements can be collected.

For input fields, COBWEB follows an approach similar to Sapienz [12] and extracts statically-defined values from the source code and layout files. Additionally, developers can provide a list of inputs, e.g., list of cities for MyTracker. Alternatively, the input values can be provided to COBWEB through symbolic execution of the app, using one of the many tools available for this purpose (e.g., [34], [35], [36], [37]).

B. Genetic Operators

We now provide a more detailed explanation of the three genetic operators in Algorithm 1.

1) Selection Operator: COBWEB implements a fitness proportionate selection strategy, a.k.a., roulette wheel selection, for breeding the next generation. That is, the likelihood of selecting a chromosome is proportional to its fitness value. The intuition behind this selection strategy is that tests that are closer to covering energy-greedy APIs or exercise them under previously unexplored contexts—thus having a higher fitness value—should have a higher chance of selection. COBWEB sorts chromosomes based on their fitness value and selects a subset of them, denoted as $P_{offspring}$, for inclusion in the next generation. The size of selected chromosomes is determined by $breedSize$ variable that is an input to the algorithm. If $F(i)$ is the fitness value for a chromosome i in the current population with size n, the probability of this chromosome to be selected for breeding is computed as follows:

$$p(i) = \frac{F(i)}{\sum_{j=1}^{n} F(j)} \quad (1)$$

2) Convergence Operator: The goal of convergence operator is to pull the population towards local optima, i.e., generate new chromosomes that largely inherit the genetic makeup of their parents. The convergence operator only changes the execution context of tests. That is, from the parents identified by the selection operator, $P_{offspring}$, it chooses those that have reached energy-greedy APIs, then uses LSM and HSM models or mocking to create a new context for those tests. The intuition behind this operator is shown in Figure 6. LSM and HSM models have finite states, thereby their search space—identified by dashed circle in Figure 6—is relatively small compared to the typical search space associated with the functional behavior of a program, represented by CTG and CG models. Convergence operator, denoted by the orange arrow in Figure 6, promotes exploration of the search space within close proximity of parent chromosomes, thereby aids the algorithm to converge to local optima.

For each chromosome in $P_{offspring}$, COBWEB randomly selects a gene to modify its context by inserting proper events in the chromosome event sequence. To avoid bloated populations, COBWEB applies convergence operator if the gene has events associated with lifecycle callbacks or hardware-related APIs. COBWEB uses two types of convergence operator: *lifecycle context operator* and *hardware context operator*.

Lifecycle context operator: To show the necessity of lifecycle context and usage of LSM for test generation, consider the second energy defect for MyTracker app described in Section II. Recall that to effectively detect this bug, a test needs to put the `TrackingActivity` into the *paused* state to assess utilization of GPS hardware in this state. To generate such test, lifecycle context operator determines current lifecycle state of the chromosome that utilizes GPS in one of its genes, and inserts the proper lifecycle callback event based on the next possible state determined from LSM.

Consider Sequence 2 of Figure 2 to see how lifecycle context operator works. The `onLocationChanged` event in `TrackingActivity` gene indicates access to GPS hardware. COBWEB realizes the lifecycle state of `TrackingActivity` is *Running* based on the last lifecycle callback in the event sequence. The next eligible state for `TrackingActivity` is *Paused* based on LSM, which can be reached by executing `onPause()` lifecycle callback. Additionally, since proper execution of a test requires the component to be in the *Running* state, COBWEB needs to include a callback to restore the component to the running state to avoid generation of invalid tests. Thereby, COBWEB generates a new chromosome corresponding to Sequence 2 of Figure 7. The input argument of `onPause` indicates that during the execution of this test, `TrackingActivity` remains in the paused state for 10 seconds.

Hardware context operator: Many energy defects manifest themselves under specific hardware settings [3], making it important to test an app under different hardware states. Recall "fail to check for connectivity" energy defect in MyTracker described in Section II. To find this energy defect, MyTracker should be tested both when there is a network connection available and not. For each chromosome in $P_{offspring}$, hardware context operator finds a gene that utilizes hardware, if any, determines the next hardware state based on the last explored state in HSM, and inserts a specific hardware state sub-gene right *before* the sub-gene that is a callback or contains APIs that utilize a hardware element.

Sequence 1: <**MainActivity**<onCreate, onClick("Download Maps")>,
MapSearchActivity<onCreate, enterText("Ottawa"), onClick("Find by Internet")>,
InternetService<onStartCommand, searchOnServer, startService("DownloadService")>,
DownloadService<onStartCommand, Utilized_Full, startDownload, onDownloadComplete>>

Sequence 2: <**MainActivity**<onCreate, onItemClick("Montreal")>,
LocationActivity<onCreate, enterText("airport"), enterText("conference"), onClick("Start")>,
TrackingActivity<onCreate, onLocationChanged("location1"), onPause("10"), onResume(), onClick("Stop")>>

Sequence 3: <**MainActivity**<onCreate, onClick("Download Maps")>,
MapSearchActivity<onCreate, onClick("Find by GPS")>,
GPSService<onStartCommand, getLastKnownLocation, startService("DownloadService")>,
DownloadService<onStartCommand, startDownload, onDownloadComplete>>

Fig. 7: Evolved event sequences from illustrative example

For example, consider a chromosome represented by Sequence 1 in Figure 2. The `startDownload` sub-gene inside the `DownloadService` gene makes an app connect to a server and download the map of Ottawa. If no prior hardware context operator is applied on `DownloadService`, the state of network would be *Disconnected* based on the Network HSM presented in Figure 4. Hence, COBWEB randomly chooses to transfer the state to either *Scanning*, *Utilized Poor*, or *Utilized Full*. Supposing the next state is chosen to be *Utilized Full*, COBWEB changes this event sequence to Sequence 1 in Figure 7. Unlike lifecycle context operator, there is no need to restore the state of hardware. That is, if a test crashes by changing the hardware state, developer has failed to properly handle that situation.

3) Divergence Operator: In contrast to convergence operator, the goal of divergence operator is to bring the population out of local optima to discover potentially better solutions, i.e., find solutions that cover new energy-greedy APIs not previously covered by tests in the current population. The intuition behind this operator is shown in Figure 6. Unlike convergence operators that perform a neighborhood search, divergence operator, denoted by the dashed green arrow, causes exploration of the whole new areas of the search space.

The goal of this operator is to explore new paths, specifically paths that cover energy-greedy APIs. To that end, it combines two operations, namely *breakup* and *reconcile* to breed a new chromosome. For each chromosome in $P_{offspring}$, breakup operation breaks it into two set of genes, passes the first set to reconcile operation, and discards the seconds set. Note that the breakup point is selected randomly and could also be the end of the chromosome, i.e., the first set is the entire chromosome and the second set is empty. At the next step, reconcile operation creates a new individual from the broken chromosome. Starting from the last gene of the broken chromosome, reconcile operation uses the CTG and CG models to generate a sequence of events that cover a path toward their leaf nodes. The operator selects a path based on a priority value. Given the following path, $\langle C_i\langle e_1,\ldots,e_{p_i}\rangle,\ldots,C_m\langle e1,\ldots,e_{p_m}\rangle\rangle$, its priority value is calculated as follows:

$$PR_{i,m} = \sum_{j=i}^{m} API_j \qquad\qquad API_j = \sum_{k=0}^{l} w_k \tag{2}$$

where API_j is a weighted sum of the number of energy-greedy APIs, l, that might be invoked during the execution of event sequences in component C_j. COBWEB takes a list of $38,626$ energy-greedy APIs from our empirical study described in Section IV-C, and counts the number of their invocations for each component using a conventional *use-def* static analysis. Since energy-greediness of APIs vary, COBWEB employs a weighted sum. To obtain the weight of each energy-greedy API, COBWEB relies on a prior study [38] that has ranked energy-greedy APIs based on their energy-greediness to compute w_k in Equation 2.

Reconcile operation may need to change the sub-genes of the last gene in the broken chromosome to reduce the likelihood of generating invalid tests. For example, consider Sequence 1 in Figure 2, where breakup operation divides it into ⟨`MainActivity`, `MapSearchActivity`⟩ and ⟨`InternetService`, `DownloadService`⟩ sequences of components. Referring to the CTG of MyTracker shown in Figure 4, reconcile chooses ⟨`GPSService`, `DownloadService`⟩ to create a new chromosome ⟨`MainActivity`, `MapSearchActivity`, `GPSService`, `DownloadService`⟩. Without changing the event sequences of `MapSearchActivity`, the test corresponding to this new chromosome would fail, as clicking on the "Find by Internet" button does not instantiate `GPSService`. Thereby, COBWEB changes the genetic makeup of `MapSearchActivity` and generates a new chromosome corresponding to Sequence 3 shown in Figure 7.

C. Fitness Evaluation

The fitness function rewards tests based on two criteria: (C_1) how close they are to covering energy-greedy APIs; and (C_2) how well they contribute to exercising different contextual factors. The first criterion is measured using CTG and CG, while the second criterion is measured using LSM and HSM.

COBWEB calculates the fitness value in two steps. First, it computes the fitness of t_i with respect to each energy-greedy API j. Then, it averages those values to compute a single fitness value for test. For each test t_i, COBWEB computes the fitness value as follows:

$$F(i) = \frac{1}{n} \times \sum_{j=1}^{n} f_i(j) \tag{3}$$

where n is the number of energy-greedy APIs on the path of t_i to a leaf in CTG and $f_i(j)$ is the fitness value of t_i with respect to energy-greedy API j, calculated as follows:

$$f_i(j) = \begin{cases} \frac{1}{3} \times [c_{1_i}(j) + c_{2_i}(j)] & \text{API } j \text{ is on the path to a leaf} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Here, $c_{1_i}(j)$ determines the fitness of t_i with respect to fitness criteria C_1. It computes how close test t_i is to cover energy-greedy API j. It is calculated as $\frac{x}{y}$, where x is the number of edges in CG to the node that contains API j, starting from the last node covered by t_i, and y is the total number of edges from root to the node that contains API j. The intuition behind this formulation is that, a test may not cover energy-greedy APIs in the early iterations. However, if it comes close to covering energy-greedy APIs, it is likely to be able to eventually cover those APIs in future iterations. Thereby, tests that exercise paths that contain more energy-greedy APIs or get close to covering such APIs should have a higher priority to evolve. If a test covers API j, $c_{1_i}(j)$ attains a value of 1.

$c_{2_i}(j)$ corresponds to fitness criterion C_2 and determines how well t_i exercises lifecycle and hardware state contexts:

$$c_{2_i}(j) = \lfloor c_{1_i}(j) \rfloor \times [l_i(j) + h_i(j)] \tag{5}$$

Here, $l_i(j)$ and $h_i(j)$ are indicators of how well t_i exercises the lifecycles of a software component and different states of a hardware element that implements API j, respectively. COBWEB computes $l_i(j)$ and $h_i(j)$ values as follows:

$$\begin{cases} 1 & \text{if the test achieve prime path coverage} \\ \frac{z}{q} & \text{otherwise} \end{cases} \tag{6}$$

where z is the length of path covered in LSM/HSM, and q is the length of the longest *prime path* for LSM/HSM. This formulation enables tests that exercise more states in LSM/HSM models to have a higher fitness value. Since execution context matters only if an API is covered by a test, Equation 5 has a coefficient $\lfloor c_{1_i}(j) \rfloor$, such that it is 0, when t_i has not reached API j, and 1, otherwise. Unless $c_{1_i}(j)$ equals to 1, the value of $\lfloor c_{1_i}(j) \rfloor$, hence $c_{2_i}(j)$, is 0 and the execution context does not matter in calculation of fitness. Finally, note that coefficient $^1/_3$ in Formula 4 is to ensure that the fitness value is between 0 and 1.

D. Test-Suite Minimization

To minimize the size of test suite, COBWEB removes tests that are subset of others, as they are unlikely to find new defects. COBWEB uses *Lowest Common Ancestor (LCA)* algorithm to find tests corresponding to overlapping paths in the graph and removes the shortest tests from T_R. For two tests $t_1 = \langle C_1, \cdots, C_m \rangle$ and $t_2 = \langle C_1, \cdots, C_n \rangle$, if the LCA between C_m and C_n is either of these nodes, these tests are likely to be overlapping. The algorithm then checks the events inside overlapping components and if they are the same, it removes the shorter test and keeps the longer one. In addition, COBWEB removes tests that fail to cover any energy-greedy APIs, as such tests are unlikely to have a significant impact on energy. Finally, the reduced test suite is transformed to Espresso tests, which can be executed on a mobile device.

VI. EVALUATION

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

RQ1. *API and execution context coverage*: How well do the generated tests cover energy-greedy APIs and exercise different lifecycle and hardware state contexts?

RQ2. *Effectiveness*: How effective are the generated tests in revealing energy defects in real-world Android apps?

RQ3. *Necessity of the models*: To what extent does using the LSM and HMS models and considering the execution context aid COBWEB to find energy defects?

RQ4. *Energy defects coverage*: What types of energy defects can be detected by COBWEB and not other energy analysis tools?

RQ5. *Performance*: How long does it take to generate tests using COBWEB?

A. Experimental Setup

Alternative Approaches: For a thorough evaluation of COBWEB, we compare it with other testing tools as well as a variety of other energy analysis approaches targeting Android. We compare COBWEB against Monkey [39], since (1) it is arguably the most widely used automated testing tool for Android, and (2) in practice, it has shown to outperform other academic test generation tools [4]. We also compare against the most recent publicly available Android testing tool, Stoat [13], shown to be superior to prior testing tools. Stoat uses a combination of model-based stochastic exploration of a GUI model of an app and randomly injected system-level events to maximize code coverage.

Subject Apps: To evaluate effectiveness of COBWEB, we needed Android apps with *real* energy defects. To eliminate any bias toward selection of subject apps in favor of COBWEB, we looked at the dataset of 8 related approaches presented in Table II and used two criteria in selecting apps. First, the energy defects identified by the approach should be confirmed by the developers of studied subject apps through a commit in the repository. Second, information about the faulty version of an app or pointers to a commit fixing the issue should be publicly available. These criteria are required to ensure the defects reported by those tools are in fact reproducible in our experimental setup and do not impose a threat to the validity of our results. From the total of $2,035$ apps studied in related approaches, only 25 matched our inclusion criteria. From those apps, we were able to reproduce the faults in 18 of them, mostly because a subset of faults in those apps related to older versions of Android and could not be reproduced in Android 6.0 that we used in our evaluation. Out of these 18 apps, we removed 3, since Soot was not able to generate complete call graphs for them. Table I shows information about our 15 subjects with real energy defects.

Fault Reproduction: To ensure the energy issues are reproducible, we executed each defective subject app under the documented use-case known to exhibit the defect. We profiled the state of hardware elements during and after execution of the app using *Trepn* [40]. Trepn is a profiling tool developed by *Qualcomm* that collects the exact power consumption data from sensors embedded in the chipset. If the profiled data indicated over-utilization of a hardware element during the execution of use-case, we marked the energy defect to be reproducible. For example, if the energy defect to reproduce is categorized as a *location defect*, we monitored the state of GPS to see if the GPS hardware is released after the execution.

B. RQ1: API and Execution Context Coverage

The objective of COBWEB is to maximize the coverage of energy-greedy APIs under various execution contexts. To evaluate the extent to which COBWEB achieves this goal, we measured API, LSM, and HSM coverage of test suites produced for our subjects. Similarly, we calculated these metrics for Monkey and Stoat as an alternative testing approach. We collected coverage information of the subjects using EMMA [41] during test execution. We ran Stoat for 3 hours, similar to the configuration used by its authors [13]. Monkey is shown to converge very close to its highest coverage at around 10 minutes [4]. However, we ran it for 1 hour to ensure sufficient testing budget. During 1 hour, it generates over $100,000$ events per subject, which is significantly higher than the $7,630$ events generated on average by COBWEB in our experiments. Table I illustrates the result of this experiment under *Coverage* column. We observe that:

COBWEB achieves a higher API coverage compared to alternative approaches. COBWEB achieves 79% API cover-

TABLE I: Subject apps and coverage information for COBWEB and alternative approaches.

Apps	Version	# Tests			Coverage									Detection				
					Energy-Greedy APIs			LSM			HSM							
		O (events)	∼L	∼H	O	M	S	O	M	S	O	M	S	O	∼L	∼H	M	S
a2dp.Vol	8624c4f	2234 (15796)	35049	22435	86%	36%	23%	96%	22%	36%	86%	0%	0%	Y	Y	Y	N	N
	b9a5768	1681 (18383)	27111	17398	87%	33%	23%	96%	22%	36%	76%	0%	0%	N	N	N	N	N
	8231d4d	1612 (22824)	27036	17674	90%	35%	26%	96%	22%	36%	78%	0%	0%	Y	Y	Y	N	Y*
	4767d64	1836 (20849)	29195	18775	88%	36%	23%	96%	22%	36%	76%	0%	0%	Y	Y	N	N	N*
GTalk	dce8b85	586 (2507)	47669	62759	94%	49%	49%	53%	14%	25%	56%	0%	0%	Y	Y	N	N	N
	c0f8fa2	531 (4836)	45406	59611	94%	51%	49%	53%	14%	25%	51%	0%	0%	Y	Y	Y	N	Y*
	5ce2d94	466 (3558)	44748	59323	94%	49%	49%	53%	14%	25%	53%	0%	0%	Y	N	N	N	N
Openbmap	56c3a67	751 (2328)	4933	2786	90%	46%	62%	80%	28%	38%	77%	0%	0%	Y	Y	N	N	N
	14d166f	746 (2984)	5343	3060	89%	45%	62%	80%	28%	38%	83%	0%	0%	Y	N	Y	N*	N*
	f72421f	754 (2980)	5410	3153	96%	46%	62%	80%	28%	38%	78%	0%	0%	Y	Y	N	N	N
OpenCamera	1.0	606 (3916)	72241	54296	33%	49%	54%	100%	26%	66%	66%	0%	0%	Y	Y	Y	Y*	Y*
Senorium	e153fdf	96 (288)	354	1127	63%	37%	56%	100%	30%	51%	86%	0%	0%	Y	N	Y	N	N
	94c9a8d	99 (336)	394	1145	63%	36%	56%	100%	30%	51%	96%	0%	0%	Y	N	Y	N	N
	94c9a8d	105 (337)	428	1360	63%	37%	57%	100%	30%	51%	85%	0%	0%	Y	N	Y	N*	N
Ushahidi	4f20612	3519 (12523)	4865	5032	79%	46%	39%	86%	59%	41%	59%	0%	0%	Y	Y	Y	Y*	Y*

O: Original COBWEB, ∼L: COBWEB without LSM, ∼H: COBWEB without HSM, M: Monkey, S: Stoat

age on average, ranging from 33% to 96% with the median of 89%. In contrast, Monkey and Stoat are able to cover on average 42% and 46% of energy-greedy APIs.

COBWEB is more effective in exercising different execution contexts compared to Monkey. While COBWEB achieves an average of 85% in covering prime paths of LSMs, ranging from 53% to 100% with the median of 96%, Monkey and Stoat are able to cover only 27% and 40% LSM prime paths on average. Alternative approaches perform worse in terms of HSM coverage, failing to cover even a single HSM prime path. This is due to the fact that neither Monkey nor Stoat are capable of effectively manipulating hardware and systematically create system events during testing.

C. RQ2: Effectiveness

We investigated the ability of COBWEB, Monkey, and Stoat for finding the energy defects in the subject apps. To that end, we executed the generated tests on a Google Nexus 6 device, running Android version 6.0. During the execution of each test, Trepn was running in the background to profile the states of hardware elements during and after execution of each test. We used the results of fault reproduction (recall Section VI-A) as our oracle. Similar to prior work [3], if the energy traces obtained during the fault reproduction and test execution matched, we determined that the test suite was able to detect the corresponding fault. Column *Detection* in Table I demonstrates the result of this study. These results show that:

Random GUI exploration and random system event injection proves to be highly ineffective. Monkey and Stoat were able to detect only 2 and 4 energy defects, respectively. The root cause of this weakness comes from their inability to cover energy-greedy APIs under different execution contexts. In fact, Monkey and Stoat were able to cover the code related to 4 and 5 energy defects, respectively—those marked with asterisk under *Detection* column. Even when covered by these tools, manifestation of those defects requires the apps to be executed under specific component lifecycle or hardware states.

COBWEB is effective for detecting energy defects. From the total of 15 verified energy defects, COBWEB was able to detect 14, where 10 of them could be detected by exercising different component lifecycle states and 4 of them could be revealed under specific hardware states. COBWEB was not able to find 1 energy defect in *a2dp.Vol*. Further investigation showed that manifestation of this energy defect requires complex interactions with the app. In fact, *a2dp.Vol* requires a user to connect a Bluetooth device to her phone, change her location, save her location in a database, and disconnect the Bluetooth device from her phone. COBWEB generated a test for each of these use-cases, but not a single test to reproduce the whole scenario, as they cover different branches of CTG.

D. RQ3: Necessity of the Models

To evaluate necessity and usefulness of LSM and HSM models, we first compared the size of test suites originally generated by COBWEB that considers these models with that generated by a modified version of Algorithm 1 that *exhaustively* injects lifecycle or hardware related events into event sequences, i.e., changed the convergence operator. In addition, we compared the ability of test suites originally generated by COBWEB in finding energy defects with that generated without using the models, i.e., we removed the consideration of execution context from the test generation process. From the results presented in Table I, we can observe that:

Contextual models make energy testing scalable. Without a model, each component of app should be exhaustively tested under all possible lifecycle/hardware states. Columns *∼L* and *∼H* under *#Tests* show the size of test suites generate by exhaustively injecting lifecycle/hardware related events to explore all possible states. We can see that by using LSM and HSM models, COBWEB is able to generate test suites that are 27 and 28 times smaller, respectively.

Execution context is crucial for detecting energy defects. Columns *∼L* and *∼H* under *Detection* illustrate the number of faults that can be detected by test suites not using either LSM or HSM models. Test suites generated without using LSM and HSM models can only detect 9 energy defects, thereby are inferior to those generated by COBWEB in terms of their ability to find energy defects. These results confirm our intuition about the importance of considering contextual conditions for energy testing.

E. RQ4: Energy Defects Coverage

We evaluated COBWEB's ability to find different types of energy defect by comparing it with the state-of-the-art energy analysis approaches. To that end, we used a recently published energy defect model for Android [3], consisting of 28 energy

TABLE II: Comparing ability of energy analysis tools to find different types of energy defects.

	Defect Model	COBWEB	[42]	[43]	[29]	[44]	[45]	[20]	[46]	[47]
Analysis Type	-	Hybrid	Static	Hybrid	Static	Static	Static	Dynamic	Dynamic	Static
Lifecycle Context	-	Y	N	N	Y	N	Y	Y	N	N
Hardware Context	-	Y	N	N	N	N	N	N	N	N
Bluetooth	3	3	0	0	0	0	0	0	2	0
Display	4	3	0	0	0	0	1	0	1	1
Location	4	2	0	1	0	1	0	1	1	0
Network	6	5	0	1	0	1	0	0	1	0
Recurring Callback	5	3	1	0	0	0	0	0	2	0
Sensor	2	2	0	1	0	1	0	1	2	0
Wakelock	4	4	0	2	2	2	0	2	3	0
Total	28	22	1	5	2	5	1	4	12	1

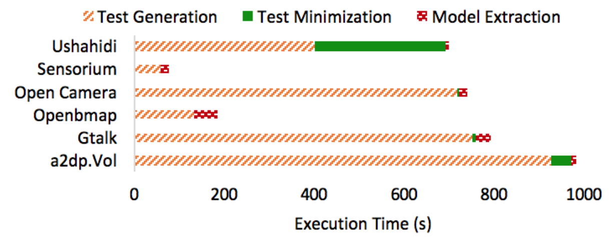

Fig. 8: Performance characteristics of COBWEB

defect types, categorized into seven groups, namely bluetooth, display, location, network, recurring callback, sensor, and wakelock. For approaches that are either not publicly available or do not work on newer versions of Android, we rely on the corresponding paper, i.e., description of the approach and limitations stated in the paper, to determine if it is able to detect each type of defect. Table II shows how these approaches differ in terms of their ability to find various types of energy defect.

We can see that **COBWEB is able to detect a wider range of energy defects compared to prior techniques**. Furthermore, it appears that dynamic analysis solutions, such as COBWEB and [46], are able to detect a wider variety of energy defects compared to static analysis solutions.

F. RQ5: Performance

To answer this research question, we evaluated the time required for COBWEB to extract models as well as the time required for test generation and test minimization. To evaluate test generation time, we measured time from when the algorithm starts generating initial population to when it terminates the loop in Algorithm 1 at Line 4. We ran the experiments on a laptop with 2.2 GHz Intel Core i7 processor and 16 GB DDR3 RAM. Figure 8 shows the performance characteristics of COBWEB for each subject app (results are averaged over various faulty versions of apps presented in Table I). From this data, we can see that COBWEB takes 23 seconds on average to extract models, 8 minutes for test generation and execution (including calculation of fitness value), and 57 seconds for test-suite minimization. These results corroborate scalability of COBWEB for test generation, making it a reasonably efficient testing tool for detecting energy issues.

VII. RELATED WORK

We provide an overview of the related research on mobile testing and green software engineering.

Mobile Testing: Test input generation techniques for Android apps mainly focus on either fuzzing to generate Intents or exercising an Android app through its GUI [4]. Several approaches generate Intents with null payloads or by randomly generating payloads for Intents [48], [49], [50], [51]. Dynodroid [16] and Monkey [39] generate test inputs using random input values. Several techniques [52], [53], [54], [55], [56], [13], [14], [15] rely on a model of the GUI, usually constructed dynamically and non-systematically, leading to unexplored program states. POLARIZ [57] uses information from crowd-based testing to enhance mobile test generation. Another set of techniques employ systematic exploration of an app in the construction of test cases: EvoDroid [10] and Sapienz [12] employ an evolutionary algorithm; ACTEve [34], JPF-Android [35], Collider [36], and SIG-Droid [37] utilize symbolic execution. Another group of techniques focus on testing for specific defects [58], [59], [17].

None of the aforementioned solutions can be used to properly test the energy behavior of Android apps, as they lack the ability to generate tests meant to exercise contextual factors.

Green Software Engineering: In recent years, several automated approaches for analysis [60], [47], [20], [38], [61], [62], [63], [64], [42], testing [46], [65], [3], [66], re-factoring [67], [68], and repair [69], [45] of mobile apps have been proposed to help developers produce more energy efficient apps.

The closest approaches to COBWEB are that of Banerjee et al. [46], GreenDroid [20], and EnergyPatch [43]. Banerjee et al. [46] present 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 (incomplete) generated graph, and explore event traces that may possibly reach energy hotspots, while profiling energy consumption of the device. The profiling process always starts from the root activity of an app, making it infeasible to test particular sequences of the app's lifecycle. Finally, the usage of a power measurement hardware makes their approach device dependent and impractical. EnergyPatch [43] fixes the scalability issue of the prior work [46] by using abstract interpretation-based program analysis to detect resource leaks instead of power trace oracle. 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. GreenDroid uses only bounded symbolic execution for finding event sequences that lead to resource leaks.

None of these techniques consider system inputs that are independent of GUI, nor do they incorporate lifecycle and hardware contextual factors in the generation of tests. They also do not generate reproducible tests. More importantly, they generate tests specifically targeted for resource leaks, failing to detect wide range of other energy defects shown in Table II.

VIII. Conclusion and Future Work

Energy efficiency is an important quality attribute for mobile apps. Naturally, prior to releasing apps, developers need to test them for energy defects. Yet, there is a lack of practical tools and techniques for energy testing. In this paper, we presented COBWEB, a search-based energy testing framework for Android. The approach employs a set of novel 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. Additionally, COBWEB implements novel genetic operators tailored to the generation of energy tests. Our experience with COBWEB on Android apps with real energy defects corroborate its ability to effectively generate useful tests to find energy defects in a scalable fashion.

Currently, we are considering several directions for future work. First, test generation is not complete without accounting for the test oracle. We are planning to explore automated methods of generating energy test oracles in future. We also plan to extend the approach for multi-objective test generation, making COBWEB a more general Android testing tool. COBWEB and research artifacts are available publicly [33].

References

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

[2] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 237–248.

[3] R. Jabbarvand and S. Malek, "μdroid: an energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 208–219.

[4] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 429–440.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[6] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[7] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.

[8] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[9] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 204–217.

[10] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.

[11] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 559–570.

[12] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[13] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 245–256.

[14] C. Zhang, H. Cheng, E. Tang, X. Chen, L. Bu, and X. Li, "Sketch-guided gui test generation for mobile applications," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 38–43.

[15] W. Song, X. Qian, and J. Huang, "Ehbdroid: beyond gui testing for android applications," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 27–37.

[16] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[17] L. L. Zhang, C.-J. M. Liang, Y. Liu, and E. Chen, "Systematically testing background services of mobile apps," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 4–15.

[18] "MyTracker Android App," 2017. [Online]. Available: https://github.com/ReyhanJB/MyTracker

[19] "Location manager strategies," 2017. [Online]. Available: https://developer.android.com/guide/topics/location/strategies.html

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

[21] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[22] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*. Springer, 2012, pp. 1–59.

[23] E. Cantu-Paz and D. E. Goldberg, "Efficient parallel genetic algorithms: theory and practice," *Computer methods in applied mechanics and engineering*, vol. 186, no. 2-4, pp. 221–238, 2000.

[24] F. Asadi, G. Antoniol, and Y.-G. Gueheneuc, "Concept location with genetic algorithms: A comparison of four distributed architectures," in *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*. IEEE, 2010, pp. 153–162.

[25] "Robolectric," 2017. [Online]. Available: http://robolectric.org/

[26] "Android testing support library : Espresso," 2017. [Online]. Available: https://google.github.io/android-testing-support-library/docs/espresso/

[27] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[28] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 77–88.

[29] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," pp. 396–409, 2016.

[30] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 29–42.

[31] "Android api reference," 2017. [Online]. Available: https://developer.android.com/reference/packages.html

[32] "crawler4j," 2017. [Online]. Available: https://github.com/yasserg/crawler4j

[33] "Cobweb website," 2018. [Online]. Available: https://sites.google.com/view/icse-cobweb

[34] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

[35] H. van der Merwe, B. van der Merwe, and W. Visser, "Execution and property specifications for jpf-android," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–5, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2557833.2560576

[36] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483777

[37] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 461–471.

[38] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 2–11.

[39] "UI/Application Excersizer Monkey," 2017. [Online]. Available: http://developer.android.com/tools/help/monkey.html

[40] L. Ben-Zur, "Using Trepn Profiler for Power-Efficient Apps," https://developer.qualcomm.com/blog/developer-tool-spotlight-using-trepn-profiler-power-efficient-apps, 2017.

[41] "EMMA: a free Java code coverage tool," http://emma.sourceforge.net.

[42] Y. Lyu, D. Li, and W. G. Halfond, "Remove rats from your code: automated optimization of resource inefficient database writes for mobile applications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 310–321.

[43] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "Energypatch: Repairing resource leaks to improve energy-efficiency of android apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 470–490, 2018.

[44] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for android apps." *IEEE Trans. Software Eng.*, vol. 42, no. 11, pp. 1054–1076, 2016.

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

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

[47] D. Li, A. H. Tran, and W. G. Halfond, "Making web applications more energy efficient for oled smartphones," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 527–538.

[48] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, ser. MoMM '13. New York, NY, USA: ACM, 2013, pp. 68:68–68:74. [Online]. Available: http://doi.acm.org/10.1145/2536853.2536881

[49] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting Capability Leaks of Android Applications," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 531–536. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590316

[50] R. Sasnauskas and J. Regehr, "Intent Fuzzer: Crafting Intents of Death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, ser. WODA+PERTEA 2014. New York, NY, USA: ACM, 2014, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/2632168.2632169

[51] A. Maji, F. Arshad, S. Bagchi, and J. Rellermeyer, "An empirical study of the robustness of Inter-component Communication in Android," in *2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2012, pp. 1–12.

[52] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[53] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *Software, IEEE*, vol. 32, no. 5, pp. 53–59, Sept 2015.

[54] W. Yang, M. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, V. Cortellessa and D. Varr, Eds. Springer Berlin Heidelberg, 2013, vol. 7793, pp. 250–265. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37057-1_19

[55] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," *SIGPLAN Not.*, vol. 48, no. 10, pp. 641–660, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2544173.2509549

[56] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 204–217. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594390

[57] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 16–26.

[58] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 118–128. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771800

[59] A. Sadeghi, R. Jabbarvand, and S. Malek, "Patdroid: permission-aware gui testing of android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 220–232.

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

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

[62] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "Ecodroid: An approach for energy-based ranking of android apps," in *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. IEEE Press, 2015, pp. 8–14.

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

[64] S. Chowdhury, S. Di Nardo, A. Hindle, and Z. M. J. Jiang, "An exploratory study on assessing the energy impact of logging on android applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1422–1456, 2018.

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

[66] H. Wu, Y. Wang, and A. Rountev, "S entinel: generating gui tests for android sensor leaks," in *Proceedings of the 13th International Workshop on Automation of Software Test*. ACM, 2018, pp. 27–33.

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

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

[69] D. Li, Y. Lyu, J. Gui, and W. G. Halfond, "Automated energy optimization of http requests for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 249–260.