Energy-Aware Test-Suite Minimization for Android Apps

Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, Sam Malek
Department of Informatics
University of California, Irvine
Irvine, CA, USA
{jabbarvr, alirezs1, hamidb, malek}@uci.edu

ABSTRACT

The rising popularity of mobile apps deployed on battery-constrained devices has motivated the need for effective energy-aware testing techniques. Energy testing is generally more labor intensive and expensive than functional testing, as tests need to be executed in the deployment environment and specialized equipment needs to be used to collect energy measurements. Currently, there is a dearth of automatic mobile testing techniques that consider energy as a program property of interest. This paper presents an energy-aware test-suite minimization approach to significantly reduce the number of tests needed to effectively test the energy properties of an Android app. It relies on an energy-aware coverage criterion that indicates the degree to which energy-greedy segments of a program are tested. We describe and evaluate two complementary algorithms for test-suite minimization. Experiments over test suites provided for real-world apps have corroborated our ability to reduce the test suite size by 84% on average, while maintaining the effectiveness of test suite in revealing the great majority of energy bugs.

CCS Concepts

•Software and its engineering → Software defect analysis; Software testing and debugging;

Keywords

Test-suite minimization, Coverage criterion, Android, Green software engineering

1. INTRODUCTION

Mobile apps have expanded into every aspect of our modern life. As the apps deployed on mobile devices continue to grow in size and complexity, resource constraints pose an ever-increasing challenge. Specifically, energy is the most demanding and at the same time a limited resource in battery-constrained mobile devices. The improper usage of energy-consuming hardware components, such as Wifi and GPS, or recurring constructs, such as loops and callbacks, can drastically drain the battery, directly affecting the usability of the mobile device [1, 5, 14].

Recent studies [20, 34] have shown energy consumption of apps to be a major concern for end users. In spite of that, many apps are abound with energy bugs, as testing the energy behavior of mobile apps is challenging. To determine the energy issues in a mobile app, a developer needs to execute a set of tests that cover energy-greedy parts of the program. This is particularly a challenge when apps are constantly evolving, as new features are added, and old ones are revised or altogether removed.

Energy testing is generally more time consuming and labor intensive than functional testing. To collect accurate energy measurements, tests often need to be executed in the deployment environment (e.g., physical mobile device), while the great majority of conventional testing can occur on capacious development environments (e.g., device emulator running on desktop or cloud). With automated mobile testing tools still in their infancy, developers spend a significant amount of their time manually executing such tests and collecting the energy measurements. The fragmentation of mobile devices, particularly for Android, further exacerbates the situation, as developers have to repeat this process for each supported platform. Thus, there is an increasing demand for reducing the number of tests needed to detect energy issues of evolving mobile software.

Prior research efforts have proposed various test-suite management techniques, such as test-suite minimization, test case selection, and test case prioritization, to help developers effectively assess the quality of software. The great majority of prior techniques have focused on the functional requirements (e.g., structural coverage and fault detection capability), and to a lesser extent non-functional requirements. Even among the the work focusing on non-functional properties, there is a dearth of prior work to account for energy issues.

In this paper, we present and evaluate a novel, fully-automated energy-aware test-suite minimization approach to determine the minimum set of tests appropriate for assessing energy properties of Android apps. The approach relies on a coverage criterion, called *eCoverage*, that indicates the degree to which energy-greedy parts of a program are covered by a test case. We solve the energy-aware test-suite minimization problem in two complementary ways. We first model it as an *integer programming (IP)* problem, which can be solved optimally with a conventional IP solver. Since the energy-aware test-suite minimization problem is NP-hard,

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...
http://dx.doi.org/10.1145/2931037.2931067

solving the integer programming model when there are many test cases is computationally prohibitive. We thus propose an approximate *greedy* algorithm that efficiently finds the near-optimal solution.

This paper makes the following contributions:

- To the best of our knowledge, the first attempt at test-suite minimization that considers energy as a program property of interest;
- An energy-aware metric for assessing the quality of test cases in revealing the energy properties of the system under test without the need for specialized power measurement hardware;
- A novel suite of *energy-aware mutation operators* that are derived from known energy bugs, in order to evaluate the effectiveness of a test suite for revealing energy bugs;
- Empirical evaluation of the proposed approach over test suites for real-world apps, corroborating the ability to reduce the size of test suites by 84%, on average, while maintaining a comparable effectiveness of original test suite for assessing the energy properties of Android apps and revealing energy bugs.

The remainder of this paper is organized as follows. Section 2 provides a background on energy issues in Android apps and motivates our work. Section 3 introduces and formulates the energy-aware test-suite minimization problem. Section 4 provides an overview of our approach, and Sections 5- 6 describe the details of our coverage metric and the minimization techniques. Section 7 presents the implementation and evaluation of the research. Finally, the paper outlines related research and concludes with a discussion of future work.

2. BACKGROUND AND MOTIVATION

Energy bugs are the main cause of battery drainage on mobile and wearable devices. They are essentially faults in the program that cause the device to consume high amounts of energy, or prevent the device from becoming idle, even when there is no user activity. Energy bugs in Android apps can be categorized as follows:

- *Wakelock bugs*—acquire an energy-greedy hardware component and fail to release it. Three major wakelock bugs are *Wi-Fi wakelock bug* [7], *CPU wakelock bug* [3], and *Sensor wakelock bug* [5]. Wakelock bugs prevent a device from becoming idle and can drastically drain its battery [14].
- *Recurring callback bugs*—are high frequency background services (for example fine-grained location update) that can consume a high amount of energy [1,2, 4,6].
- *Loop bugs*—are high frequency loops that repeatedly utilize an energy-greedy component such as Wi-Fi, thereby rapidly increase the energy consumption of an app.

Figure 1 presents an example of such bugs inspired by those found in real-world Android apps. The code snippet

```
1  Wifilock lock = ((WifiManager) this.getSystemService()).createWifiLock();
2  lock.acquire();
3  for (int i = 0; i < X; i++) {
4      URL url = new URL(resourceLink[i]);
5      HttpURLConnection conn = url.openConnection();
6      conn.connect();
7      file = downloadFile(conn.getInputStream());
8      processFile(file);
9      in.close();
10 }
```

Figure 1: Code snippet with energy bugs.

depicts a loop that accesses and downloads X files from a list of servers (line 4–7), processes them (line 8), and closes the connection (line 9). Before starting the loop, the code acquires a lock on the Wi-Fi resource (line 2) to prevent the phone from going into stand-by during download. This implementation can result in both the wakelock and loop bugs. Studies have shown that network components can remain in a high power state, even after a routine has completed [29,30]. Such a state is referred to as *tail energy.* Tail energy is not an energy bug itself, but interleaving a network related code and a CPU-intensive code in a loop can exacerbate its impact and cause energy bug.

To perform energy testing and find possible energy bugs, a developer should design test cases that cover *energy-greedy* segments of the program—segments that contribute more to the energy cost of the app—and measure the energy consumption of device during execution of those test cases. Spikes in energy measurements that last long period of time as well as high energy consumption of a device without the user interacting with the device are good indicators of energy bugs [14,27].

Figure 2 shows the energy consumption trace of a Nexus 6, during the execution of a test case for the code shown in Figure 1 that downloads five files, before (solid line) and after (dashed line) fixing the mentioned energy bugs. Keeping the Wi-Fi connection open during processing the files increases the average power consumption of the device (the area under curve). Also, failing to release Wi-Fi lock keeps the device awake and the phone keeps consuming energy, even after a routine has completed. By splitting the single loop into two loops to fix the loop bug (one for downloading all the files first and one for processing them later) and releasing the Wi-Fi wakelock after downloading files to fix the wakelock bug, the average power consumption of test case is decreased and the power state of the device before and after execution of test case remains the same.

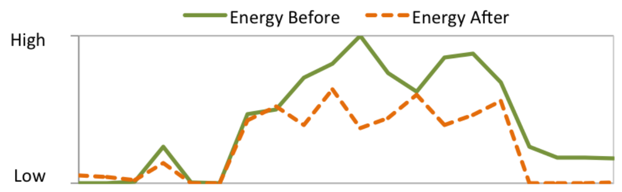

Figure 2: Energy consumption trace of a test case for the code snippet in Figure 1, before (solid line) and after (dashed line) fixing energy bugs.

Testing non-functional properties, particularly energy consumption, which is recently gaining substantial interests due to the increasing use of battery-constraint devices, is relatively under-explored compared to those aimed at functional correctness [19]. A test suite of a mobile app is adequate for energy testing, if it can effectively find all energy bugs in the code. That is, if test cases of a test suite *cover all the energy-greedy segments of the program that contribute to total energy cost of the app* under different use-cases, the test suite is adequate for energy testing. Detecting all the energy bugs in the program is not decidable. For example, for a small number of data files in Figure 1 (X), the impact of tail energy might be negligible. However, for a large number of such files, the loop bug occurs, which can rapidly drains the battery of the device. As such, deciding what values for X may result in energy bug is complicated. Testers, thus, usually settle on coverage metrics as adequacy criteria.

The commonly used coverage metric in test-suite minimization problems, statement coverage, is unable to discriminate among statements according to their energy consumption. Studies have shown that energy consumption varies significantly across bytecodes [17], lines of code [24], and system APIs [26]. That is, test cases with the same statement coverage may cover different lines and consume different amount of energy during execution. For example, the test case a, even with a lower statement coverage than the test case b, may demonstrate higher energy cost, if it executes the code that utilizes energy-greedy API calls. As a result, statement coverage is not a suitable metric for energy-aware test-suite minimization.

For an energy-aware test adequacy criterion, the energy consumption needs to be measured, estimated, or modeled for further identification of energy inefficiencies of the code [14]. Prior research proposed fine-grained approaches to either measure or estimate the energy consumption of mobile apps [24, 29]. The precise energy measurement can be used for optimizing energy usage of an application under test. However, an intuitive metric for assessing the quality of test case to identify energy-greedy segments is still missing. Moreover, most techniques require power measurement hardware to measure energy cos, which comes with technical requirements and challenges.

To overcome the limitations of structural coverage metrics, we propose a novel energy-aware coverage metric, collectively referred as *eCoverage*, that indicates the degree to which energy-greedy segments of a program are covered by a test. eCoverage discriminates among different energy-greedy segments based on their energy cost and whether they re-execute during the execution of test case.

3. ENERGY-AWARE TEST-SUITE MINIMIZATION

To clarify our proposed idea for energy-aware test-suite minimization, we formally define the problem as follows:

Given: (1) A program P consisting of p segments, $S = \{s_1, s_2, \dots, s_p\}$, with $m \leq p$ energy-greedy segments $\in S'$, to be tested for assessing energy properties of P; (2) A test suite $T = \{t_1, t_2, \dots, t_n\}$ with each test case represented as a coverage vector $\vec{V_{t_i}} = \langle v_{i,1}, \dots, v_{i,m} \rangle$, such that $v_{i,j}$ is 1 if t_i covers energy-greedy segment s_j, and 0 if t_i does not cover energy-greedy segment s_j; and (3) a non-negative function $w(t_i)$ that represents the significance of a test case in identifying energy bugs.

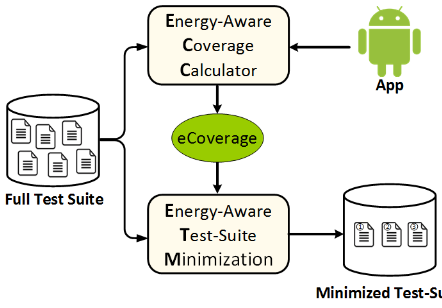
Energy-Aware Coverage Calculator
App
Full Test Suite
eCoverage
Energy-Aware Test-Suite Minimization
Minimized Test-Suite

Figure 3: Energy-aware test-suite minimization framework.

Problem: Find the smallest test suite $T' \subseteq T$, such that T' covers all energy-greedy segments covered by T, and for every other T'' that also covers all energy-greedy segments $|T'| \leq |T''|$ and $\sum_{t_i \in T'} w(t_i) \geq \sum_{t_i \in T''} w(t_i)$.

Program segments are individual units of a program, which can be defined fine-grained (e.g., statements) or coarse-grained (e.g., methods). The energy consumption of a segment depends mainly on the energy-greedy APIs invoked by that segment (e.g., *network* APIs consume more energy than *log* APIs [26]) and on recurring constructs (e.g., loops or recurring callbacks [23]). Energy-greedy segments highly contribute to the total energy consumption of the program. Therefore, a test case that covers energy-greedy segments during its execution has a higher significance for energy testing of app, compared to the one covering less greedy segments.

To reduce the risk of discarding significant test cases during test-suite minimization, we calculate the eCoverage of each test case. eCoverage takes a value between 0 and 1, and indicates the degree to which energy-greedy segments of the program are covered by a test case (more details in Section 5). The function $w(t_i) = eCoverage_{t_i}$ in problem definition allows us to characterize the significance of a test case t_i so that we select tests with the highest eCoverage.

There might be several test cases in a test suite that cover the same energy-greedy segments. Thereby, the original test suite T can be partitioned into subsets of $T_1, T_2, \dots, T_m \subseteq T$, such that any test case t_i belonging to T_j covers energy-greedy segment $s_j \in S'$. A representative set of test cases that covers all of the s_js in S' must contain at least one test case from each T_j; such a set is called the *hitting set* of $T_1, T_2, \dots, T_m$. The minimal hitting set problem is shown to be NP-hard, using a reduction to the set covering problem [32]. Our formulation of test-suite minimization is, therefore, an instance of *weighted set cover*. The original test suite might not be intended for energy testing, rather developed for functional or structural testing. As a result, the test cases in T might not cover all the energy-greedy segments, but a subset of them.

4. APPROACH OVERVIEW

Figure 3 depicts our framework for energy-aware test-suite minimization, consisting of two major components: (1) *Energy-Aware Coverage Calculator (ECC)* which is responsible to calculate the eCoverage for each test case, t_i, in the original test suite of the given app, using program analysis; and (2) *Energy-Aware Test-Suite Minimization (ETM)* component that identifies the minimum subset of test cases from T, suitable for energy testing of the given app.

Our ECC component statically analyzes an app to obtain its call graph and annotates each node of the call graph with energy cost estimates. Using the execution traces of test cases in the available test suite, the eCoverage of each test case will be calculated by mapping execution path information to the annotated call graph (Section 5).

After computing eCoverage of tests in the test suite, ETM component produces a minimized test suite suitable for energy testing, which aids a developer by reducing the effort needed to inspect the test results, especially for identifying energy bugs in the code. ETM component performs the energy-aware test-suite minimization in two complementary ways, optimal yet computationally expensive integer programming (IP) technique, and efficient near-optimal greedy approach (Section 6).

Using energy-aware test-suite minimization, the search space for assessing energy properties of the app and identifying plausible energy bugs is reduced to handful of test cases, helping the developer in fixing such issues with less effort and time. Our framework also delivers execution traces of test cases and energy estimate of executed energy-greedy segments, helping developers to understand which sequences of invoking energy-greedy segments are more energy consuming and to pinpoint root cause of energy bugs.

In the following two sections, we describe the details of the *Energy-Aware Coverage Calculator* and *Energy-Aware Test-Suite Minimization* components.

5. ENERGY-AWARE COVERAGE CALCULATOR

For the purpose of this work, we propose *eCoverage* that has the following beneficial properties: (1) it is computationally efficient to measure; (2) it can be defined at different levels of granularity (e.g., statement, API, or method levels); and (3) measuring it does not actively require the use of special monitoring hardware. We developed a hybrid static and dynamic analysis approach to calculate the eCoverage.

In this paper, we consider program segments (cf. Problem Definition in Section 3) to be methods of a program or system APIs and thereby, the definition of eCoverage is at the granularity of methods. For illustrating the concepts in this section, we use a hypothetical app whose call graph is shown in Figure 4. Each node of the call graph is a segment, and the colored nodes denote *energy-greedy segments* that highly contribute to the energy consumption of the app.

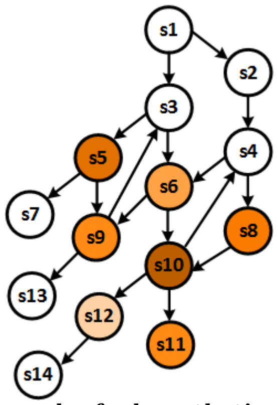

Figure 4: Call graph of a hypothetical Android app.

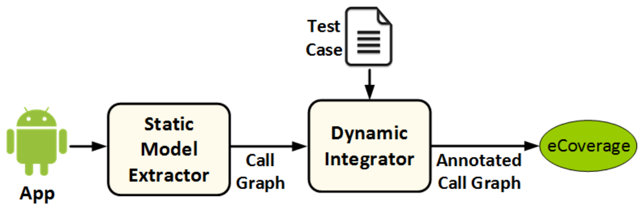

Figure 5: Overview of the ECC component.

ECC component that is responsible to calculate the eCoverage for each test case consists of two sub-components shown in Figure 5: (1) *Static Model Extractor*, which statically analyzes the app to obtain its call graph; and (2) *Dynamic Integrator*, which collects the execution trace of the input test case, maps it to the call graph, and annotates call graph segments with the energy estimates in order to compute eCoverage for the given test case.

To calculate eCoverage, *Static Model Extractor* first extracts the app's call graph and then identifies energy-greedy segments—methods with at least one system API in their implementation. For a test case t_i, each energy-greedy segment s_j (i.e., a method in an Android app) is then annotated by *Dynamic Integrator* with a *segment score*, $sc_{j,i}$, which represents the estimated amount of energy consumption by the given segment during execution of test case t_i. The segment score is calculated as $sc_{j,i} = r_j \times f_{j,i} \times \sum_{k=1}^{I_{j,i}} e_k$, where r_j denotes the structural importance of energy-greedy segment s_j in the call graph, $f_{j,i}$ represents the frequency at which energy-greedy segment s_j is invoked during execution of test case t_i, $I_{j,i}$ is the number of system APIs in the implementation of energy-greedy segment s_j invoked during execution of test t_i, and e_k is a pre-measured average energy cost for an API k.

Methods reachable along more paths in a call graph are more likely to contribute to the energy cost of the app. Thus, the *Static Model Extractor* component heuristically calculates r_j as the multiplication of its incoming and outgoing edges. If the segment is a sink (with no outgoing edge) or a source (with no incoming edge), we consider only the number of incoming or outgoing edges, respectively. For example, there are two incoming and three outgoing edges for the segment s_{10} in the call graph of Figure 4; thus $r_{10} = 6$.

To assess the values of $f_{j,i}$ and $I_{j,i}$, the *Dynamic Integrator* component records the invocation of methods and system APIs in a log file and counts the number of invocations for segment s_j and APIs inside it during execution of t_i. For e_k, our approach uses the results from [26] to supply the average energy cost of each API.

After calculating segment scores and annotating the call graph, the *Dynamic Integrator* component computes eCoverage of test case t_i as follows:

$$eCoverage_{t_i} = \frac{\sum_{j=1}^{m} sc_{j,i} \times v_{i,j}}{\sum_{j=1}^{m} \max_a\{sc_{j,1}, \ldots, sc_{j,a}\}} \tag{1}$$

where m is the number of energy-greedy segments and $v_{i,j}$ is a binary variable denoting whether test case t_i covers energy-greedy segment s_j (cf. Problem Definition). eCoverage takes a value between 0 and 1. A test with a higher eCoverage is more likely to reveal the presence of an energy bug with a substantial impact on the energy consumption of the app.

Similar to other coverage criteria, the denominator computes the *ideal* coverage that can be achieved and the numerator indicates the coverage achieved by a given test case. In our formulation of eCoverage, the numerator estimates the energy consumed by a given test and the denominator estimates the highest energy consumed in each energy-greedy segment, considering all the test cases in the test suite. That is, the denominator is the maximum segment score estimated by test cases that cover the energy-greedy segment s_j, for all a test cases that cover it.

6. ENERGY-AWARE TEST-SUITE MINIMIZATION

In this section, we describe two approaches to perform energy-aware test-suite minimization for Android apps. The first one leverages integer programming, IP, to model the problem, and the second one is a greedy algorithm. Our proposed approaches aim to determine the minimum set of tests appropriate for assessing energy properties of Android apps and find possible energy bugs in the program.

6.1 Integer Non-linear Programming

The energy-aware test-suite minimization problem can be represented as an IP model consisting of (1) decision variables, (2) an objective function, and (3) a constraint system.

6.1.1 Decision Variables

We let the binary variable t_i represent the decision of whether a test case appears in the minimized test-suite or not. That is, a value of 1 for t_i indicates that the minimized test-suite includes the corresponding test, while a value of 0 indicates otherwise. Using boolean decision variables, the minimized test suite can be represented as an array of binary values $\langle t_1, t_2, ..., t_n \rangle$, where n is the number of test cases in the original test suite.

6.1.2 Objective Function

The goal of energy-aware test-suite minimization is to reduce the size of test suite, while maintaining the ability of the original test suite to assess energy properties of the app and reveal energy bugs. To achieve this goal, test cases in the minimized test suite should cover all the energy-greedy segments of the program that are covered by the original test suite. In addition, tests should be distinguished according to their ability in identifying energy bugs to avoid discarding important test cases during minimization. To find such a subset of the original test suite, we formulate the objective function as follows:

$$min \sum_{i=1}^{n} (1 - eCoverage_{t_i}) \times t_i \qquad (2)$$

where n is the number of test cases in the original test suite. Definition of objective with a minimum function ensures that the solution is the smallest subset of original test suite. Since $eCoverage_{t_i}$ value for a test takes a value between 0 and 1, a test with high eCoverage has low value for $1 - eCoverage_{t_i}$. Thereby, weighing test cases by the coefficient $1 - eCoverage_{t_i}$ ensures selection of significant test cases such that $\sum_{t_i \in T'} eCoverage_{t_i} \geq \sum_{t_i \in T''} eCoverage_{t_i}$ for any other subset $T'' \subseteq T$ with $|T'| \leq |T''|$ (cf. problem definition in Section 3). By replacing the formula of $eCoverage_{t_i}$ from the Equation 1, the objective function can be re-written as follows:

$$min \sum_{i=1}^{n} \left(1 - \frac{\sum_{j=1}^{m} sc_{j,i} \times v_{i,j}}{\sum_{j=1}^{m} \max_a\{sc_{j,1}, \ldots, sc_{j,a}\}}\right) \times t_i \qquad (3)$$

To achieve the optimal solution, the model should select a test case that covers the largest number of energy-greedy segments not covered by the previously selected tests. Unlike code coverage metrics, where a test case contributes to the coverage by covering a statement or a branch only once, eCoverage values change depending on the number of times an energy-greedy segment is covered by tests. The complexity of criterion entails that the coverage vector of each test case, and consequently its corresponding eCoverage, in the original test suite should be updated upon each selection. That is, a test case that covers energy-greedy segments already covered by previously selected test cases is not significant anymore (i.e., not likely to reveal new energy bugs), therefore its eCoverage should be decreased.

To that end, we weigh each energy-greedy segment by $\prod_{k_j}(1 - t_{k_j})$, as shown in Formula 4, where k_j denotes the number of test cases cover energy-greedy segment s_j. If an energy-greedy segment is covered by at least one of the selected test cases, this coefficient evaluates to zero. As a result, test case that covers other uncovered energy-greedy segments has a higher chance for selection.

$$min \sum_{i=1}^{n} \left(1 - \frac{\sum_{j=1}^{m} sc_{j,i} \times v_{i,j} \times \prod_{k_j}(1 - t_{k_j})}{\sum_{j=1}^{m} \max_a\{sc_{j,1}, \ldots, sc_{j,a}\}}\right) \times t_i \qquad (4)$$

Note that due to the multiplication of decision variables in Formula 4, IP formulation of energy-aware test-suite minimization is non-linear.

6.1.3 Constraints

To ensure that the minimized test suite covers all the energy-greedy segments that are covered by the original test suite, we need to certify that each energy-greedy segment is covered by at least one of the test cases in the minimized test suite. Such constraints can be encoded in the IP model as follows:

$$\sum_{i=1}^{n} v_{i,j} \times t_i \geq 1 \quad (1 \leq j \leq m) \qquad (5)$$

where m denotes the number of energy-greedy segments and n is the available test cases in the original test suite. The jth constraint in Formula 5, thus, ensures that at least one of the test cases covering the energy-greedy segment s_j will be in the minimized test suite. The model does not require constraints on other segments, since the right hand of the constraint is 0, which makes the constraint trivial.

6.2 Integer Linear Programming

There is no known algorithm for solving an integer non-linear programming (INLP) problem optimally other than trying every possible selection. Furthermore, for problems with non-convex functions, IP solvers are not guaranteed to find a solution [35]. For all of these reasons, we needed to investigate other options to solve the energy-aware test-suite minimization problem.

We have leveraged a technique for transforming the above non-linear problem into a linear one by adding new *auxiliary* variables, $v'_{i,j}$, defined as $v_{i,j} \times \prod_{k_j}(1 - t_{k_j}) \times t_i$. As a result, $v'_{i,j}$ takes a value of 1, if the test case t_i covers the energy-greedy segment s_j that is not covered by the previously selected test cases, and 0 otherwise. The value of 1 for $v'_{i,j}$ stipulates that the test case t_i covers the energy-greedy segment s_j, which is not covered by previously selected test cases. If a test case does not cover s_j, or if it is covered by the selected test cases, $v'_{i,j}$ takes the value 0. Using auxiliary variables $v'_{i,j}$, the objective function of our IP model from Formula 4 can be rewritten as follows:

$$min \sum_{i=1}^{n} (t_i - \frac{\sum_{j=1}^{m} sc_{j,i} \times v'_{i,j}}{\sum_{j=1}^{m} \max_a\{sc_{j,1}, \ldots, sc_{j,a}\}}) \qquad (6)$$

In addition to the adjustment of objective function, we need to introduce additional constraints to control the auxiliary variables. To ensure that $v'_{i,j}$ equals to 1 for energy-greedy segments not previously covered by selected test cases and equals to 0 otherwise, we add the following set of constraints to the model:

$$v'_{i,j} \leq t_i \quad (\forall\ s_j \text{ covered by } t_i) \qquad (7)$$

$$\sum_{i=1}^{n} v_{i,j} \times v'_{i,j} = 1 \quad (1 \leq j \leq m) \qquad (8)$$

According to Formula 7, if a test case t_i is not selected, then $v'_{i,j}$ takes a value of 0. On the other hand, if the test case t_i is selected, the variable $v'_{i,j}$ can take a value of either 0 (if energy-greedy segment s_j is covered by the previously selected test cases) or 1 (if s_j is not covered by the previously selected test cases). The constraint in the Equation 8 entails that if s_j is covered by one of the selected test cases, the value of $v'_{i,j}$ for any test case t_i which is not selected yet being set to 0.

The use of auxiliary variables allows us to remove the multiplication of decision variables from the objective function. However, this transformation significantly increases the complexity of the problem, which in turn makes it computationally expensive. The high complexity of ILP approach for large-size problems motivated us to devise additional algorithms.

6.3 Greedy Algorithm

Algorithm 1 outlines the heuristic, energy-aware test-suite minimization process. It takes the original test suite generated for an Android app under test as input, and provides a minimized set of test cases as output. The algorithm first iterates over tests in the test suite and computes coverage vector (line 5) of tests as well as the coverage vector of original test suite (line 6). It then selects the test case with highest

Algorithm 1: Greedy Algorithm for Energy-Aware Test-Suite Minimization

```
   Input: $T$ Original Test Suite
   Output: $T'$ Minimized Test Suite with the same eCoverage
1  $T' \leftarrow \{\}$;
2  $\vec{V_T} \leftarrow \vec{0}$;
3  $\vec{V_{T'}} \leftarrow \vec{0}$;
4  foreach $t_i \in T$ do
5      $\vec{V_{t_i}} \leftarrow getCoverageInfo(t_i)$;
6      $\vec{V_T} \leftarrow \vec{V_T} \vee \vec{V_{t_i}}$;
7  while $\vec{V_{T'}} \neq \vec{V_T}$ do
8      $findMax(t_i s \in \{T - T'\})$ based on $t_i.eCoverage$;
9      $t_i \leftarrow removeMax(T)$;
10     $T' \leftarrow T' \cup \{t_i\}$;
11     $\vec{V_{T'}} \leftarrow \vec{V_{T'}} \vee \vec{V_{t_i}}$;
12     foreach $t_i \in T - T'$ do
13         $reCalculate(\vec{V_{t_i}}, t_i.eCoverage)$;
```

eCoverage that covers energy-greedy segments not yet covered by previously selected tests (lines 8–11). Afterwards, the algorithm updates the coverage vector and eCoverage value of the remaining tests in the original test suite (lines 12–13). This greedy process then repeats until selected test cases cover all the energy-greedy segments that are initially covered by the original test suite.

To make the idea concrete, consider Table 1 that illustrates the algorithm through five test cases t_1, t_2, t_3, t_4, and t_5 for our running hypothetical app, whose call graph is shown in Figure 4. Each inner table represents the coverage vector for the test cases—sorted according to their eCoverage for a better comprehension—and the coverage vector of the minimized test suite ($\vec{V_{T'}}$) at one iteration of the algorithm. The first iteration selects t_5 (covering segments s_1, s_8, s_{10}, and s_{11}) as the test with the highest eCoverage.

Algorithm 1 then updates the coverage vector of the remaining test cases. Table 1 shows the updated coverage information for the test suite in iteration 2 and after the selection of t_5. Since t_5 is already selected, the segments covered by t_5 are no longer considered in calculating eCoverage of remaining tests. Only the energy-greedy segments that have not been covered by t_5 (s_5, s_6, s_9, and s_{12}) are included. Test case t_1 is then selected at the end of iteration 2. This process repeats until selected test cases in the minimized test suite cover all the energy-greedy segments covered by the original test suite. In this example, the greedy approach selects t_5, t_1, t_2, and t_3 as the minimized test suite after four iterations.

The example illustrates the point that the greedy algorithm can result in sub-optimal solutions. While the ILP-based approach solves this problem with three test cases, t_1, t_3, and t_5, to cover all the energy-greedy segments, the greedy strategy selects four test cases. This is mainly due to the fact that the greedy algorithm starts from the test case with the greatest eCoverage, which may lead to a local optimum solution.

7. EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of our proposed framework for energy-aware test-suite minimization. Our evaluation addresses the following questions:

Table 1: Running example for the greedy algorithm

Iteration 1: t_5 is selected

Tests	s_5	s_6	s_8	s_9	s_{10}	s_{11}	s_{12}	eCov
t_2	0	1	0	1	0	0	0	0.42
t_1	1	0	0	1	0	0	0	0.53
t_3	0	1	0	0	1	0	1	0.54
t_4	0	0	1	0	1	0	1	0.63
t_5	0	0	1	0	1	1	0	0.69
$\vec{V_T'}$	0	0	1	0	1	1	0	

Iteration 2: t_1 is selected

Tests	s_5	s_6	s_8	s_9	s_{10}	s_{11}	s_{12}	eCov
t_4	0	0	0	0	0	0	1	0.01
t_3	0	1	0	0	0	0	1	0.04
t_2	0	1	0	1	0	0	0	0.12
t_1	1	0	0	1	0	0	0	0.23
t_5								
$\vec{V_T'}$	1	0	1	1	1	1	0	

Iteration 3: t_2 is selected

Tests	s_5	s_6	s_8	s_9	s_{10}	s_{11}	s_{12}	eCov
t_4	0	0	0	0	0	0	1	0.01
t_3	0	1	0	0	0	0	1	0.04
t_2	0	1	0	0	0	0	0	0.05
t_1								
t_5								
$\vec{V_T'}$	1	1	1	1	1	1	0	

Iteration 4: t_3 is selected

Tests	s_5	s_6	s_8	s_9	s_{10}	s_{11}	s_{12}	eCov
t_4	0	0	0	0	0	0	1	0.01
t_3	0	0	0	0	0	0	1	0.03
t_2								
t_1								
t_5								
$\vec{V_T'}$	1	1	1	1	1	1	1	

RQ1. *Effectiveness:* How effective are our proposed techniques in reducing the size of original test suite? Is the minimized test suite as effective as the original test suite in revealing energy bugs?

RQ2. *Correlations:* What is the relationship between eCoverage and statement coverage of a test case? What is the relationship between eCoverage and energy consumption of a test case?

RQ3. *Performance:* What is the performance of our prototype tool implemented atop a static analysis framework and an IP solver? How scalable are the proposed IP and greedy algorithms?

7.1 Experiment Setup

To evaluate our proposed techniques in practice, we collected real world apps from *F-Droid*, a software repository that contains open source Android apps. We randomly selected 15 apps from different categories of F-Droid repository for evaluation.

We used test cases automatically generated using Android Monkey [11]. To that end, we ran Monkey for two hours for all apps, with configuration to generate test cases exercising 500 events (i.e., touch, motion, trackball, and system key events). We considered the test cases generated during this time as the original test suite of apps. Prior to applying optimization techniques, our framework requires obtaining eCoverage information about the test cases of subject apps. In addition to eCoverage, we collected statement coverage information using *EMMA* [8].

To statically analyze the apps for calculating eCoverage, *Static Model Extractor* (Figure 5) employs the Soot framework [13, 33] that provides the libraries for Android static program analysis. To collect the execution traces of test cases, we implemented a module using the *Xposed* framework [12] that records the invocation of methods and system APIs in a log file, which is later processed to extract information about the executed paths in each app.

In calculating eCoverage, we rely on the average energy consumption of system APIs, e_k, measured by Linares and collegaues [26]. These e_k values are obtained by manually utilizing and running 50 popular apps on Google Play several times, and is the average of energy consumption of each API in different scenarios. The energy consumption of APIs might change depending on the context and the device the apps are running on. Considering more devices and context only require additional pre-measured values as an input to our ECC component, but does not impose any change to the approach.

We used lp-solve [9], an open source mixed integer linear programming solver, to solve the IP models, and ran the experiments on a computer with 2.2 GHz Intel Core i7 processor and 16 GB DDR3 RAM. The input files for the solver are automatically generated using the coverage information provided by the ECC component. Our research artifacts are available for download [10].

7.2 RQ1: Effectiveness

To evaluate the effectiveness of our minimization techniques, we compared the percentage of reduction in size of original test suite for the subject apps. Additionally, we assessed the impact of reduction on the effectiveness of the original test suite in revealing energy bugs. To assess the effectiveness of test suite, we developed a novel form of mutation analysis for energy testing according to known energy bugs in Android apps, as outlined in Section 2.

Table 2 shows examples of our energy-aware mutation operators. For *wakelock mutants*, we created the mutants by injecting the mutation operators in proper parts of the code. For example, we created Wi-Fi wakelock mutants by adding the *acquire* API before the code that is responsible to download object(s). If the app already used Wi-Fi wakelock, we commented out the release API on *onPause* and *onDestroy* methods. For *expensive background service mutants*, we changed the arguments of the mentioned methods in Table 2 to a smaller values so that the periodic task executed at a higher rate during execution of test cases. For

Table 2: List of major energy bugs in Android apps as fault model and corresponding energy-aware mutation operators

Energy Bug Type	Mutation Operator Examples
Wi-Fi wakelock	((WifiManager) getSystemService(Context.WIFI_SERVICE)).createWifiLock().acquire()
CPU wakelock	((PowerManager) getSystemService(Context.POWER_SERVICE)).newWakeLock().acquire()
Sensor wakelock	((SensorManager)getSystemService(SENSOR_SERVICE)).getDefaultSensor(Sensor_type)
Recurring callback	Timer.schedule(period), Timer.scheduleAtFixedRate(period)
Recurring callback	ScheduledThreadPoolExecutor.scheduleAtFixedRate(period),
Recurring callback	AlarmManager.setRepeating(intervalMillis), AlarmManager.setInExactRepeating(intervalMillis)
Loop	Java loop constructs e.g., while and for

Table 3: Effectiveness of energy-aware test-suite minimization approaches in reducing the size of test-suite and maintaining the ability to reveal energy bugs

					IP		Greedy	
Apps	#Tests	LoC	#Methods	%Killed	Reduction	%Killed	Reduction	%Killed
Apollo	150	20,520	1,691	46%	72%	33%	66%	38%
Open Camera	106	15,064	1,035	57%	76%	55%	69%	51%
Jamendo	183	8,709	749	100%	70%	85%	68%	85%
Lightning Browser	100	7,219	427	65%	84%	62%	81%	62%
L9Droid	189	7,458	446	75%	86%	75%	83%	75%
A2DP Volume	130	6,670	395	43%	80%	43%	77%	43%
Blockinger	124	3,924	276	56%	76%	56%	72%	56%
App Tracker	221	3,346	291	50%	88%	50%	85%	50%
Sensorium	229	3,288	259	75%	93%	75%	92%	75%
Androidomatic	156	2,156	91	100%	83%	100%	83%	100%
AndroFish	250	1,499	109	53%	88%	51%	88%	48%
SandwichRoulette	233	1,443	129	100%	87%	100%	86%	100%
anDOF	224	1,176	108	74%	91%	74%	89%	61%
AndroidRun	100	1,021	53	100%	85%	100%	84%	100%
Acrylic Paint	200	936	61	68%	94%	68%	92%	68%
Average	-	-	-	71%	84%	68%	81%	67%

expensive loop mutants, we increased the number of iterations whenever possible, thereby the loop in mutant version executed more times than the original version of app.

To determine whether an energy mutant is killed, we measured the energy consumption of the tests using Trepn [15]. We experienced that the energy consumption level of the device on the post-run phase—after the execution of test is completed—is higher than the pre-run phase—before the execution of test—in most of the mutants (recall Figure 2). Since this pattern was not seen among all the wakelock mutants, we monitored the active system calls to kernel related to the wakelocks, before and after the execution of test cases. As a result, a test case kills a wakelock mutant if the number of wakelocks after the execution of test case is more than the number of wakelocks before it. For expensive background service and expensive loop mutants, our measurements demonstrated that a test case kills the mutant, if the average energy consumption of test case during the execution of mutant is higher than that of the original version.

Table 3 shows the number of tests in the original test suite of subject apps (column 2) and the percentage of mutants killed by the tests in the original test suite (column 5), as well as percentage of reduction by each proposed minimization approach (column 6 and 8 for IP and greedy, respectively) and the percentage of mutants killed by the tests in the minimized test suites (column 7 and 9 for IP and greedy, respectively). These results demonstrate that we can on average reduce the size of a given test suite by 84% using IP approach and 81% using greedy approach, with a negligible penalty of loosing effectiveness of the test suite by 3% and 4% using IP and greedy, respectively.

As expected, IP achieves a greater test reduction than greedy in all cases, corroborating that the solutions produced by IP are in fact optimal. For the majority of subject apps, both IP and greedy kill the same number of mutants. In *Apollo* app, however, the greedy algorithm achieves a higher ratio of killed mutants compared to the IP approach. This can be attributed to two factors: (1) The greedy approach does not reduce the number of tests as much as IP, thus, the higher number of killed mutants can be due to the fact that more tests are executed in the case of greedy. (2) eCoverage is only an estimate for evaluating the quality of tests for revealing energy properties of the software. Any discrepancy between eCoverage and the actual energy cost of executing a test can prevent the IP and greedy algorithms from picking the best tests, i.e., tests that kill the mutants.

7.3 RQ2: Correlations

To demonstrate the need for a new coverage metric for energy testing, we examined the correlation between eCoverage and statement coverage, as well as its correlation with energy consumption. Statement coverage is commonly used as an adequacy metric in test-suite minimization. As a result, we compared the correlation of eCoverage with statement

Table 4: Pearson Correlation Coefficient (r) of <eCoverage, statement coverage> and <eCoverage, energy cost> series for subject apps

Apps	$r_{statement\ coverage}$	$r_{energy\ cost}$
Apollo	0.21	0.94
Open Camera	0.2	0.57
Jamendo	0.89	0.93
Lightning Browser	-0.11	0.99
L9Droid	0.5	0.92
A2DP Volume	0.43	0.82
Blockinger	0.86	0.94
App Tracker	0.35	0.85
Sensorium	0.37	0.72
Androidomatic	0.52	0.85
AndroFish	0.34	0.95
SandwichRoulette	0.59	0.81
anDOF	0.41	0.94
AndroidRun	0.17	0.69
Acrylic Paint	0.1	0.75

coverage to assess the extent in which statement coverage can be substituted for eCoverage.

To that end, we calculated the Pearson correlation coefficient for two series of $\langle eCoverage, statementcoverage\rangle$ and $\langle eCoverage, energyconsumption\rangle$. We estimated the energy cost of each test case similar to [21], by aggregating the average energy cost of all system APIs invoked during execution of test case. Pearson correlation coefficient, a.k.a. Pearson's r correlation, measures the linear relationship between two variables and takes a value between -1 and +1 inclusive. A value of 1 indicates positive correlation, 0 indicates no relation, and -1 indicates negative correlation. More precisely [31], absolute value of r between 0 to 0.3 stipulates no or negligible relationship, between 0.3 to 0.5 indicates weak relationship, between 0.5 to 0.7 indicates moderate relationship, and higher than that indicates strong relationship.

The results on Pearson correlation coefficient—denoted by r—of 2,255 test cases for subject apps are shown in Table 4. r values in Table 4 indicate that there is almost a negligible or weak correlation between eCoverage and statement coverage of subject apps. On the other hand, eCoverage holds a strong correlation with the actual energy cost of a test case, confirming eCoverage to be a proper metric for energy testing. We noticed that for two of the subject apps, Jamendo and Blockinger, the correlation between statement coverage and eCoverage is strong. Our manual investigation shows that the majority of statements in the implementation of these two apps are system APIs. As a result, the overlap between covered statements and covered APIs are high, thereby eCoverage is correlated to statement coverage.

7.4 RQ3: Performance

In this section, we evaluate the performance of different elements of our approach (recall Figure 3).

7.4.1 *Energy-Aware Coverage Calculator*

Energy-aware coverage calculator, ECC, consists of two sub-components, *Static Model Extractor* and *Dynamic Integrator*. To calculate eCoverage of tests for an app, we need to extract the app's call graph, and then map the execution paths of each test case to the call graph. Figure 6 presents

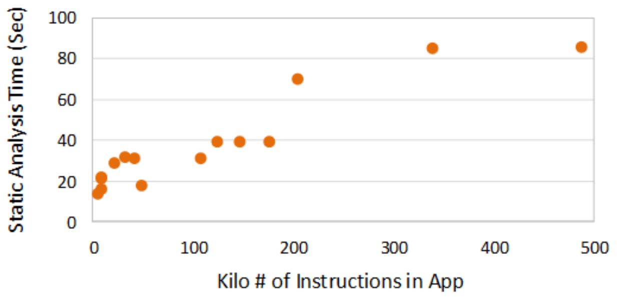

Figure 6: Performance of Static Model Extractor.

the time taken by the Static Model Extractor to extract call graphs of the subject apps. The scatter plot shows both the analysis time and the app size in kilo number of instructions. According to the results, our approach analyzes 80% of subject apps in less than one minute to extract their models, with the overall average of 38 seconds per app.

The performance analyses on test suite of subject apps show that the time taken by the *Dynamic Integrator* component to calculate eCoverage of tests in the full test suite is negligible, 2 seconds on average for all subject apps. Our approach leverages Xposed for run-time instrumentation of the root Android process, rather than instrumentation of an app's implementation. The execution time overhead incurred using Xposed to collect execution paths of test cases is 7.36%±1.22% on average with 95% confidence interval.

7.4.2 *Energy-Aware Test-Suite Minimization*

To compare the performance of techniques for energy-aware test-suite minimization proposed in this paper, we measured the execution time of each approach. Our evaluation results indicate that the greedy approach takes less than a second, 14.2 ± 10.3 milliseconds on average with 95% confidence interval, to solve the minimization problem. The execution of the IP approach on the other hand, takes between 1 second to 7 hours, to minimize test suites of different subject apps. We observed that the execution time of the IP approach heavily depends on the characteristics of the problem, e.g., the number of constraints (bounded by the size of test suite × number of energy-greedy segments) and decision variables (the size of test suite).

Figure 7 shows the sensitivity of IP approach for the five subject apps whose execution time takes more than an hour. The IP formulation of these apps for their original test suite consists of over 10,000 constraints. To generate the graph,

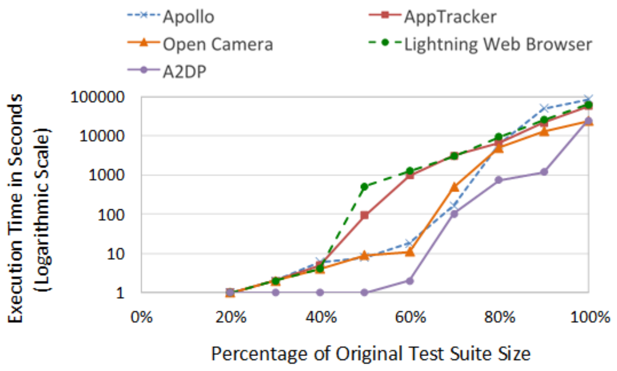

Figure 7: Sensitivity of execution time of integer programming approach to the size of test suite.

we gradually increased the set of tests included from the full test suite of these subject apps. We repeated the experiments for 30 times to ensure the confidence interval of 95% on the average execution time values. It can be seen that execution time of the IP approach for each app increases logarithmically, as the size of test suite—number of decision variables—grows linearly.

These results confirm that the greedy algorithm demonstrates better performance than IP, and is more scalable to larger problems. However, the IP approach is optimal and results in test suites with smaller size. As a result, test suites generated by the IP approach consume less energy and save the developer's time.

8. RELATED WORK

Our energy-aware test-suite minimization is related to prior work on test-suite maintenance as well as approaches for measuring and estimating the energy consumption.

Test-Suite Maintenance: Previous work in test-suite maintenance can be categorized as *test-suite minimization*, *test case selection*, *test case prioritization*, and *test-suite augmentation* techniques [36]. These approaches are categorized to subcategories in the literature [16] as: (1) *Random*, which selects arbitrary number of available test cases in an ad hoc manner; (2) *Retest all*, which naively reruns all the available test cases; (3) *Coverage-based data flow*, which selects test cases that exercise data interactions that have been affected by modifications; and (4) *Safe selection*, which selects every test case that achieves a certain criterion.

The majority of these techniques consider adequacy metrics for functional requirements of the test suite and to lesser extent non-functional requirements [28]. To the best of our knowledge, there are only few approaches in the literature that perform test-suite maintenance with respect to the energy consumption.

Kan [22] investigated the use of Dynamic Voltage and Frequency Scaling (DVFS) for energy efficiency during regression testing. This work focuses on the assumption that over the versions of a program that do not have significant changes in functionality, CPU-bound tests remain CPU-bound, and similarly IO-bound tests remain IO-bound. It is effective therefore, to optimize the processor frequency for the execution of CPU-bound test to achieve a good level of energy savings. Unlike our approach, which is a test-suite minimization, this work utilizes *retest all* [16] technique and re-runs all the existing tests. In addition, the goal of this work is to reduce the energy consumption of whole test suite, rather than selecting tests that are good for energy testing.

Another closely related work is an energy-directed approach for test-suite minimization [25]. The proposed approach in these papers tries to generate energy-efficient test suites that can be used to perform post-deployment testing on embedded systems. To that end, the authors measured the energy consumption of test cases, using a hardware, and used those information to perform test-suite minimization. This approach is not suitable for energy testing, since it discards tests with high energy consumption, which are necessary for detecting energy inefficiencies.

Unlike the aforementioned techniques, which focus on running the test cases in the most energy-efficient way, our approach selects the minimum subset of the existing test suite that can be used for energy testing of the Android apps. Our approach is complementary; an interesting avenue of future research is a combined multi-objective approach, where both the energy cost of running the tests and their ability to reveal energy defects are considered in the selection of tests.

Energy Consumption: Prior studies related to energy consumption of Android apps can be categorized into *power modeling* and *power measurement*. Research in power modeling suggests estimating the energy usage of mobile devices or apps in the absence of hardware power monitors [18, 24]. Studies in power measurement make use of specialized hardware to determine an app's energy consumption at various granularities. None of these prior studies provide test adequacy metric for determining the energy-efficiency of tests. As a result, in this paper, we proposed eCoverage as an energy-aware adequacy metric to perform energy-aware test-suite minimization.

Li and colleagues [25] proposed the use of execution time as a metric for test-suite optimization, when energy consumption information is not available. Though collecting execution time for test cases is easy, using time as a proxy for energy is controversial. Although some research shows that execution time is perceived to be positively correlated with energy consumption [23], others suggest that time is not an appropriate proxy for identifying energy-greedy segments of the program [18].

9. CONCLUSION AND FUTURE WORK

As mobile apps continue to grow in size and complexity, the need for effective testing techniques and tools that can aid developers with catching energy bugs grows. In this paper, we presented a fully-automated, energy-aware test-suite minimization approach to derive the minimum subset of available tests appropriate for energy testing of Android apps. The approach employs a novel energy-aware metric for assessing the ability of test cases in revealing energy bugs.

We described two ways of reducing the tests: an integer programming formulation that produces the optimal solution, but may take a long time to execute; and a greedy algorithm that employs heuristics to find a near-optimal solution, but runs fast. The experimental results of evaluating the two algorithms on real-world apps corroborate their ability to significantly reduce the size of test suites (on average 84% in the case of IP and 81% in the case of Greedy), while maintaining test suite quality to reveal the great majority of energy bugs. To evaluate the effectiveness of the test suites, we developed a novel suite of energy-aware mutation operators that are derived from known energy bugs.

Currently, we are considering several directions for future work. First, we plan to extend our proposed approach to other test-suite maintenance problems, i.e. energy-aware test case prioritization and energy-aware test-suite augmentation. We also plan to build on this work to devise automated test generation techniques that are targeted at executing the energy greedy segments of the program.

10. ACKNOWLEDGMENT

This work was supported in part by awards CCF-1252644 from the National Science Foundation, D11AP00282 from the Defense Advanced Research Projects Agency, W911NF-09-1-0273 from the Army Research Office, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.

11. REFERENCES

[1] Android developer website: Alarm manager. http://developer.android.com/reference/android/app/AlarmManager.html.

[2] Android developer website: Best practices for background jobs. http://developer.android.com/reference/java/util/concurrent/ScheduledExecutorService.html.

[3] Android developer website: Power manager. http://developer.android.com/reference/android/os/PowerManager.html.

[4] Android developer website: Scheduled thread pool executer. http://developer.android.com/reference/java/util/concurrent/ScheduledThreadPoolExecutor.html.

[5] Android developer website: Sensor manager. http://developer.android.com/reference/android/app/AlarmManager.html.

[6] Android developer website: Timer. http://developer.android.com/reference/java/util/Timer.html.

[7] Android developer website: Wi-fi manager. http://developer.android.com/reference/android/net/wifi/WifiManager.WifiLock.html.

[8] Emma java code coverage tool. http://emma.sourceforge.net/.

[9] lpsplve. http://lpsolve.sourceforge.net/5.5/.

[10] Project website. https://seal.ics.uci.edu/tools/etm-tool/.

[11] UI/Application Excersizer Monkey. http://developer.android.com/tools/help/monkey.html.

[12] Xposed Framework. http://repo.xposed.info/.

[13] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices* (2014), vol. 49, pp. 259–269.

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

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

[16] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM) 10*, 2 (2001), 184–208.

[17] Hao, S., Li, D., Halfond, W. G., and Govindan, R. Estimating android applications' cpu energy usage via bytecode profiling. In *The Intl. Workshop on Green and Sustainable Software* (2012), pp. 1–7.

[18] Hao, S., Li, D., Halfond, W. G., and Govindan, R. Estimating mobile application energy consumption using program analysis. In *The Intl. Conf. on Software Engineering* (2013).

[19] Harman, M., Jia, Y., and Zhang, Y. Achievements, open problems and challenges for search based software testing.

[20] Heikkinen, M. V., Nurminen, J. K., Smura, T., and Hämmäinen, H. Energy efficiency of mobile handsets: Measuring user attitudes and behavior. *The Telematics and Informatics* (2012).

[21] Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S., and Ammann, P. Ecodroid: An approach for energy-based ranking of android apps. In *Proceedings of the 4th International Workshop on Green and Sustainable Software* (2015), pp. 8–14.

[22] Kan, E. Y. Energy efficiency in testing and regression testing–a comparison of dvfs techniques. In *Quality Software (QSIC), 2013 13th International Conference on* (2013), IEEE, pp. 280–283.

[23] Li, D., Hao, S., Gui, J., and Halfond, W. G. An empirical study of the energy consumption of android applications. In *The Intl. Conf. on Software Maintenance and Evolution* (2014).

[24] Li, D., Hao, S., Halfond, W. G., and Govindan, R. Calculating source line level energy information for android applications. In *The Intl. Symposium on Software Testing and Analysis* (2013), pp. 78–89.

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

[26] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. Mining energy-greedy api usage patterns in android apps: An empirical study. In *The Working Conf. on Mining Software Repositories* (2014).

[27] Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., and Voelker, G. M. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *NSDI* (2013), vol. 13, pp. 57–70.

[28] Orso, A., and Rothermel, G. Software testing: a research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering* (2014), ACM, pp. 117–132.

[29] Pathak, A., Hu, Y. C., and Zhang, M. 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* (2012), ACM, pp. 29–42.

[30] Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y.-M. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems* (2011), EuroSys '11, pp. 153–168.

[31] Rumsey, D. J. *Statistics for dummies.* John Wiley & Sons, 2011.

[32] Sipser, M. *Introduction to the Theory of Computation, 3rd edition.* Course Technology, 2012.

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

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

[35] Wolsey, L. Integer programming. *Wiley* (1998).

[36] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability 22*, 2 (2012), 67–120.