

RESEARCH ARTICLE - METHODOLOGY

Supporting Software Evolution Through Feedback on Executing/Skipping Energy Tests for Proposed Source Code Changes

Cagri Sahin | Lori Pollock | James Clause

Computer and Information Sciences
Department, University of Delaware, Newark,
DE, USA

Correspondence

Cagri Sahin, Email: cagri@udel.edu

Abstract

With the increasing use of battery-powered devices comes the need to test mobile applications for energy consumption and energy issues. Unfortunately, energy testing is expensive because it is a manual, labor-intensive process that often requires multiple, separate, energy-measuring devices to collect energy usage data. The high costs of energy testing can negatively affect the planning process of application evolution. For example, developers might be limited in the number of changes they can include in a release because they must conservatively plan to conduct energy testing after each change.

In this paper, we present a new approach to provide developers with feedback on executing/skipping energy tests for proposed code changes. Our technique leverages change impact analysis and pre-computed API energy usage information. More specifically, for a proposed change, the technique predicts whether energy testing will be required, and if so, which energy tests will need to be run. Such information may allow developers to avoid spending unnecessary time for energy testing and develop an effective application evolution timeline. To investigate the feasibility of our technique, we implemented a prototype for Android applications and conducted three case studies at different granularity levels on ten Android applications.

KEYWORDS:

software evolution, software energy testing, repository mining, source code changes, android applications

1 | INTRODUCTION

With the increasing use of mobile and battery-powered devices, the energy consumption of applications has become a major concern for users¹. Such devices have limited battery lifetimes. As a result, energy inefficiencies in applications can cause rapid battery drain which directly affects the usability of such devices. For example, if an application acquires, but fails to release, energy consuming resources such as WIFI or GPS, these resources continue to stay in a high-power state and consume excessive amounts of energy. This can lead to users complaining about applications, writing poor reviews, deleting applications, or requesting refunds^{2,1}. To satisfy the demands and expectations of users, developers must consider the energy efficiency of their applications. Therefore, in addition to allocating resources to test for traditional requirements like correctness, developers must also allocate resources to conduct energy testing to check their applications for energy consumption issues.

Unfortunately, energy testing is often more expensive than traditional types of testing that aim to verify and validate an application's correctness. The costs of energy testing are due to its characteristic features. First, to collect accurate energy usage data during an execution, energy tests should be run on real devices with specialized energy measurement hardware. Second, since the low sampling rates of energy measurement hardware

may affect the measurements, energy tests must have long running times. Finally, energy tests have to be performed for each supported platform, which, in the case of mobile applications, can be a significant number of devices.

The high costs of energy testing can adversely impact the planning process of application evolution. Similar to traditional testing, energy testing should be performed in response to code changes, which occur frequently in mobile applications. Currently, developers plan code changes without knowing energy tests to be executed/skipped for the changes. To detect and prevent energy issues as early as possible, they must anticipate conducting energy testing after each change by running all energy tests. However, a proposed code change might not require all energy tests to be run or might not even require any energy testing at all. This lack of information prevents developers from making decisions on code changes such as ordering, postponing, or canceling them. The majority of existing work on energy testing has focused on minimizing test suites with respect to their energy consumption, investigating the adequacy of test suite for energy testing, finding energy bugs, and reducing the energy consumption of test suites^{3,4,5,6,7}. To the best of our knowledge, none of the existing work focuses on identifying the amount of energy testing required for proposed changes.

In this paper, we present a technique, Energy Retest Umpire (ERU), that provides feedback on executing/skipping energy tests for proposed code changes. This feedback can help developers plan for changes and allocate testing resources. ERU informs developers about whether energy testing is required for a proposed code change before they actually make the change. It also identifies what energy tests need to be executed when energy testing is necessary. At a high level, ERU leverages change impact analysis and pre-computed API energy usage information.

To explore ERU, we implemented a prototype for Android applications. Using the prototype, we performed a preliminary study on ten freely-available, open source Android applications. The goal of the study is to investigate the feasibility of ERU and how it performs when changes are expressed at differing granularity levels. The results of our exploratory study are promising and show that ERU is feasible and can provide useful feedback. We believe that, with further research and development, ERU has the potential to become a practical tool for planning mobile application evolution.

This paper makes the following contributions:

- (1) A new approach, ERU, that provides feedback on executing/skipping energy tests for proposed code changes.
- (2) ERU can be implementable for Android applications.
- (3) Case studies that demonstrates energy testing is not always required for a proposed single source code unit change.

The remainder of this paper is organized as follows: Section 2 introduces background information about energy testing and code evolution. Section 3 provides an overview of ERU with example scenarios. Section 4 and Section 5 present the implementation and exploratory study of the prototype of ERU, respectively. Section 6 discusses the results of the study including potential threats to its validity. Finally, Section 7 and Section 8 discuss related work and present our conclusions and future work.

2 | BACKGROUND

2.1 | Energy Testing

Software bugs that lead to energy inefficiencies in applications are known as energy bugs. These energy bugs cause excessive battery drain, which is a main user complaint about applications. Different types of energy bugs can be found in mobile applications including resource overuse and misuse bugs, no-sleep bugs, sleep conflicts bugs, loop bugs, immortality bugs, activity bugs, and event bugs^{8,9,6}. For example, no-sleep bugs keep at least one component of the mobile device awake erroneously. This prevents the mobile device from going to a lower power state and increases the battery drain significantly.

To combat energy issues in applications, developers need to perform energy testing. Energy testing differs from traditional testing in several ways. In particular, it often requires developers to execute energy tests on real devices and collect energy usage data by using a special energy measurement device. In general, the energy testing process works as follows. Developers install the application under test on a real device. Then, they run the application and execute energy tests while energy usage data is collected by the energy measurement device. The collected data is analyzed to detect energy issues. This manual, labor intensive process is then repeated several times for each supported platform.

There are three different energy measurement approaches which are hardware based, simulation based, and estimation based. To accomplish high accuracy in energy measurement, hardware based approach is more preferable than other approaches. In this approach, special energy measurement devices are used to measure actual power consumption externally and they do not introduce any measurement overhead.

Functional tests that are designed to test correctness of the applications are generally not usable as energy tests for several reasons. First, energy tests should have long running times. Functional tests such as unit tests can take time in milliseconds. This amount of time might not be enough to collect sufficient energy usage data due to the low sampling rates of energy measurement hardware. Second, they should focus

on energy bugs and features of the application that use energy-greedy hardware components in the device to detect energy issues. In addition, energy tests need to be user interaction scenarios for mobile applications since display usage should be taken into account in order to measure energy consumption of mobile applications which are interactive and event-driven. Developers can generate deterministic scenarios manually or generate random scenarios via an automatic event generator (e.g., Monkey¹⁰). These generated scenarios are recorded and reproduced by using capture/replay tools (e.g., RERAN¹¹). Manually reproducing scenarios can influence energy consumption because developers can perform the same sequence of actions (e.g., enter text into a textbox or click a button) but cannot maintain the same timing between the actions.

Since energy tests are user interaction scenarios for mobile applications, each energy test has different event and action ordering with a focus on some application features. For example, adding a note and deleting a note might be two different energy tests for Tomdroid, a note-taking application. Because energy tests focus on application features instead of individual source code units of the application, executing all energy tests that cover the modified source code unit is needed to obtain good accuracy in energy testing.

2.2 | Code Evolution

Developers frequently need to evolve their applications to add new features, enhance or adjust existing features, fix bugs, meet new requirements, improve performance, restructure the source code, etc. The process of evolving an application most likely involves code changes and testing in response to those code changes. When developers make code changes, they should perform traditional and energy testing since both correctness and energy-efficiency of the application are important. Due to lack of information about executing/skipping energy tests for the proposed code changes, developers should plan to conduct energy testing after each change and run all energy tests. This might negatively affect the application evolution timeline and restrict the number of changes that developers plan to include in a release.

2.3 | ERU in Code Evolution

Our technique, ERU, can be used whenever developers have proposed code changes as knowing energy tests to be executed/skipped for proposed code changes may allow developers to avoid unnecessary energy testing efforts. Developers can also develop a realistic and effective application evolution timeline by having such information while planning the changes. For example, some changes may not require energy testing, and thus the allocated time for energy testing can be used to add more changes in a release. When developers are up against a deadline and have to decide what code changes are made in this release, they may postpone code changes that require energy tests to a later release. They can even decide to cancel changes that require enormous amount of energy testing efforts. As a hardware-based testing environment is needed to perform energy testing, developers can order the changes to reduce setup costs. In addition, there are usually various ways to implement a feature or functionality. Comparing the necessary energy testing efforts of different implementations can also help developers to decide which choice to apply.

3 | ENERGY RETEST UMPIRE

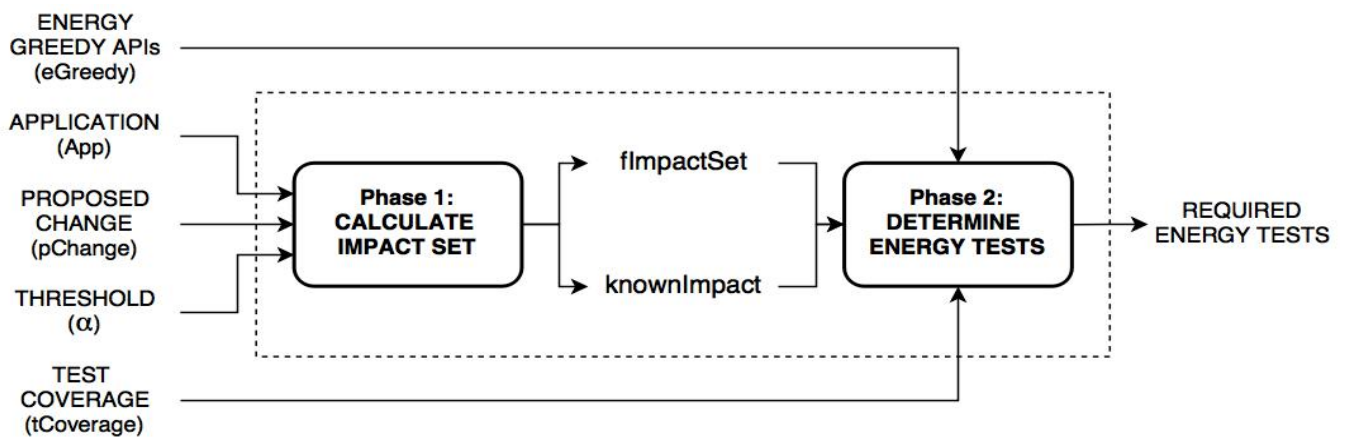


FIGURE 1 High-level overview of Energy Retest Umpire Technique.

The aim of ERU is to provide feedback on executing/skipping energy tests for proposed code changes to developers which is distinctly different from focusing on finding energy bugs or making applications more energy efficient. Figure 1 presents a high-level view of ERU. The main insight behind ERU is that energy tests to be executed/skipped for a proposed change depend on the energy greedy API usage of the proposed change and the source code units impacted by the change. ERU takes the following as input:

Application (App): Source code of an application.

Proposed Change (pChange): Potential change that the developer plans to make to the input application. It can be expressed at different source code unit granularities (e.g., package, file, method, etc.).

Energy Greedy APIs (eGreedy): API methods that are known, a priori, to consume significant amounts of energy. ERU uses a standard set, but the set is expandable.

Test Coverage (tCoverage): Coverage information indicating which application source code units (expressed at the same granularity as the proposed changes) are executed by each energy test. For example, at the method level $t_1 \mapsto \{m_1, m_2, m_3\}$ indicates that energy test 1 covers methods 1, 2, and 3.

Threshold (α): Confidence threshold used to filter the impact set of the proposed change (see Section 3.2).

ERU produces the following as output:

Required Energy Tests: A list of energy tests that need to be run for the proposed change. In general, there are three possible outcomes.

- No Energy Tests: The proposed change does not require any energy tests to be run (i.e., the list of tests is empty).
- Partial Energy Tests: The proposed change requires some energy tests be run (i.e., the list contains some, but not all, energy tests).
- All Energy Tests: The proposed change requires all energy tests to be run (i.e., the list contains all energy tests).

To decide whether energy testing is required, and if so, which energy tests need to be run, ERU is divided into two main phases: the *Calculate Impact Set* phase and the *Determine Energy Tests* phase.

The first phase, *Calculate Impact Set*, identifies the potential impacts of the proposed change on other source code units. It takes as input the application, proposed change, and threshold and generates *knownImpact*—a flag that indicates whether the impact of the proposed change is known and *fImpactSet*—the source code units that will likely be changed along with the proposed change for the given threshold. These source code units are at the same granularity with respect to the proposed change. For example, if the proposed change is an application method, then its *fImpactSet* contains the potentially impacted application methods, which are the co-changes of the proposed change.

The second phase, *Determine Energy Tests*, takes as input the outputs of the first phase as well as the Energy Greedy APIs (eGreedy) and Test Coverage (tCoverage) information. It uses these inputs to determine energy tests to be executed/skipped for the proposed change.

3.1 | Example Scenarios

As intuitive examples of how ERU determines the energy tests to be executed/skipped for a proposed change, consider the scenarios shown in Figures 2, 3 and 4. These scenarios represent the possible outputs of ERU.

In each figure, the left subfigure, (a), shows the API method calls of the application's source code units such as application's method, files, classes, etc. Nodes inside the rectangle represent application source code units and nodes outside the rectangle represent API methods. The application source code unit where the developer plans to make code changes (pChange) is solid black. The right subfigure, (b), illustrates impacted application source code units (fImpactSet), energy greedy API methods (eGreedy), and *knownImpact* flag computed by *Calculate Impact Set* and the *Determine Energy Tests* phases. Source code units in fImpactSet are lightly shaded if fImpactSet $\neq \emptyset$. API methods that are marked with a cross ('X') show the eGreedy methods that consume high energy. They can influence application energy consumption.

Figure 2 shows the scenario where pChange does not require energy testing. The impact of pChange is known as the *knownImpact* flag is set to true. It means that, depending on the use of eGreedy methods, energy testing may or may not be required. In this case, since none of the application source code units in fImpactSet call any of the eGreedy methods, no energy tests needed to be run for pChange.

Similarly to Figure 2, the impact of the pChange is known in Figure 3. The difference is that some of the source code units in fImpactSet call eGreedy methods. Therefore, energy testing is required and the energy tests that cover any of these application source code units should be run for pChange. Note that, depending on the specific coverage of the tests, they may or may not all need to be run.

Finally in Figure 4, the *knownImpact* flag is false because the impact of the proposed change is unknown. In this case, all energy tests should be run for pChange whether pChange uses eGreedy methods or not.

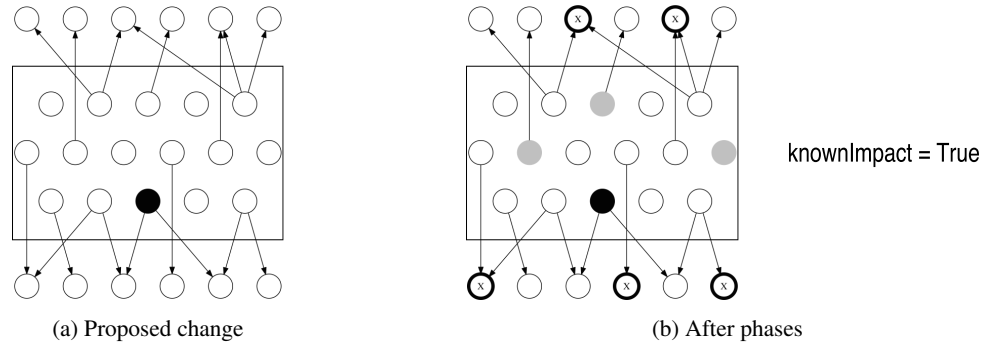


FIGURE 2 No energy tests scenario.

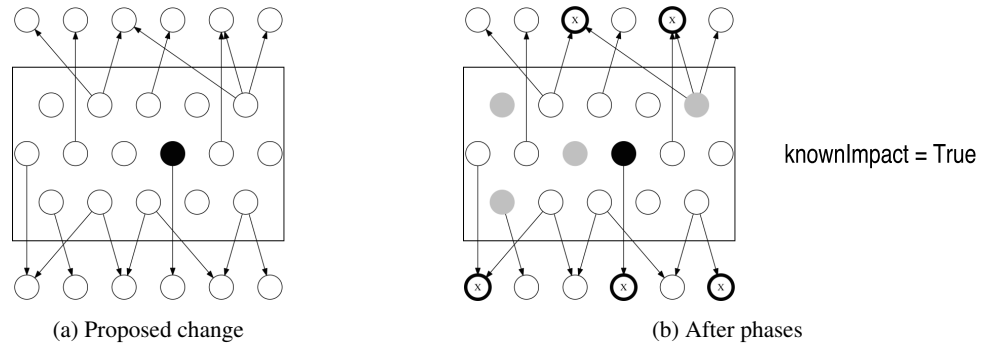


FIGURE 3 Partial or all energy tests scenario.

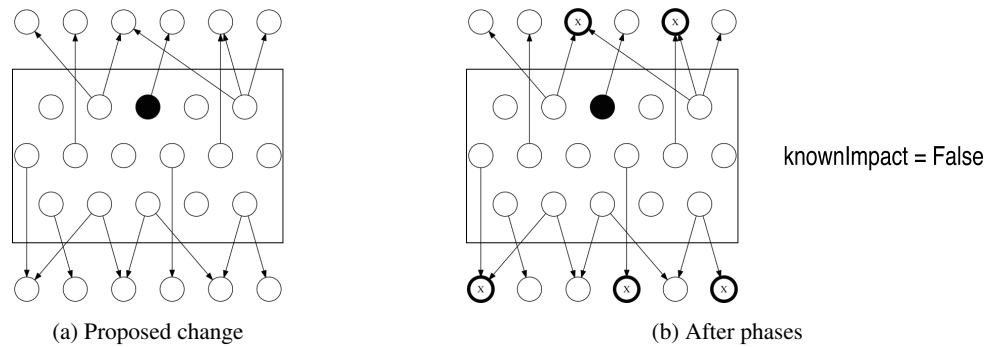


FIGURE 4 All energy tests scenario.

3.2 | Calculate Impact Set

The goal of the *Calculate Impact Set* phase is to identify the impacts of the proposed change on other application source code units. One way to achieve this goal is to use change impact analysis¹². Change impact analysis computes the impact set of the proposed change, formulated as a set of pairs $\langle n, p \rangle$, where each pair is composed of a source code unit n and the probability p of the source code unit being impacted by the proposed change. The probability ranges from 0 to 1, where 1 is the highest probability of being impacted.

The *Calculate Impact Set* phase outputs a flag called *knownImpact* that indicates whether the impact set is empty because no other changes are anticipated or because there is insufficient information to make a prediction. The flag is set to false when there is insufficient information. The other output of the *Calculate Impact Set* phase, *flmpactSet*, is a filtered version of the impact set that only includes pairs where the probability is greater than or equal to the selected threshold (α). Source code units in *flmpactSet* are considered co-changes of the proposed change that will be changed along with the proposed change. We also add the proposed change into its *flmpactSet* with the probability of 1. For example, assume A

is the proposed change, and B and C are impacted with probabilities of 0.9 and 0.8, respectively. The impact set of A is ($\langle B, 0.9 \rangle, \langle C, 0.8 \rangle$). Then, the *flmpactSet* of the proposed change becomes ($\langle A, 1.0 \rangle, \langle B, 0.9 \rangle$) for threshold 0.85 and ($\langle A, 1.0 \rangle, \langle B, 0.9 \rangle, \langle C, 0.8 \rangle$) for threshold 0.8.

3.3 | Determine Energy Tests

Determine Energy Tests phase takes as input the *flmpactSet*, *eGreedy*, *tCoverage*, and *knownImpact*. It first checks the *knownImpact* flag, and adds all energy tests to *RequiredEnergyTests* if the *knownImpact* flag is False, which means there is insufficient information. If not, all *callee* API methods of each *source code unit* in *flmpactSet* are examined. When any of the *callee* API methods is in *eGreedy*, energy tests in *tCoverage* that cover the *source code unit* are added to *RequiredEnergyTests*. This process continues until all source code units in *flmpactSet* are inspected. Finally, *RequiredEnergyTests*, which consists of energy tests that should be run for testing energy impact of the proposed change is returned as output.

4 | IMPLEMENTATION

An Android application is composed of several components: *manifests*, *source code files*, and *non-code resources*¹³. Manifests contains *AndroidManifest.xml* file that provides essential information about the application to the Android system in order to run the application. While source code files are found in “*./java/*” directory as java files, non-code resources such as user interface (UI) description and bitmap images are found in “*./res/*” directory.

This section describes the prototype implementation of ERU designed for Android applications. Android source code files, which are java files, use APIs in the Android framework. Since energy greedy APIs (*eGreedy*) is a necessary input for ERU, we used pre-computed Android API energy usage information to obtain this input. The precision of ERU relies on the precision of the underlying change impact analysis and any change impact analysis technique could be used. To generate the impact set of a proposed change (*pChange*) in *Calculate Impact Set* phase, the prototype leverages Historical Change Impact Analysis. Additionally, identifying the callees of the Android application source code units is needed in the *Determine Energy Tests* phase to determine required energy tests. Therefore, we integrated *DependencyFinder*¹⁴ into *Determine Energy Tests* phase. We provide more details about the implementation in following subsections.

4.1 | Energy Greedy APIs

To derive the set of energy greedy APIs, we made a literature review on published research papers that are related to both Android APIs and energy consumption of Android applications. We then identified APIs and individual API methods which consume significant amounts of energy from the information provided by these papers.

More specifically, the set of energy greedy APIs that we considered in our prototype is derived from two sources. First, we used the results of the findings of an empirical study conducted to find the most energy-greedy Android API methods¹⁵. This study examined the energy consumption of 55 Android apps from different domain categories by using real-usage scenarios and analyzed 807 Android API methods. Based on the results of their study, they categorized 131 methods as energy greedy.

In addition, we also considered recent studies that show *wakeLock*, *GPS*, and *GSM* related APIs can cause battery drain because of either misuses of APIs such as *wakeLock* APIs or uses of energy greedy hardware components such as *GPS* and *GSM*^{16,17,18,19}. However, these studies did not investigate API methods individually, therefore we added *wakeLock*, *GPS*, and *GSM* API packages with all of their methods to our *eGreedy* set.

4.2 | Change Impact Analysis

Software change impact analysis (CIA) estimates co-changes that need to be made to accomplish a change¹². Change impact analysis approaches use different scopes, including source code, formal models, and miscellaneous artifacts²⁰. Source-code based CIA approaches focus mainly on identifying the part of the code that needs to be modified along with the proposed change. These approaches can be static or dynamic²¹. Dynamic CIA techniques require execution of the source code to collect information after a change while static CIA techniques analyze information about the source code before a change. Source-code based CIA techniques generate impact sets at different granularity levels such as file, class, method, field, or statement. Further, both dynamic and static techniques can be divided into subtypes. Dynamic CIA can use either offline or online analyses. Static CIA can perform historical, textual, or structural static analyses.

In our prototype implementation, we chose to use historical CIA. Historical change impact analysis (HCIA) is a static, source-code based analysis that extracts co-change couplings by mining changes in the source code repository of an application. HCIA computes a confidence value with the

predicted changes. We chose to use HCIA in our implementation for several reasons. First, HCIA is the most used technique among the source-code based change impact analysis approaches²¹. Second, it does not require actually applying changes and executing application source-code to generate an impact set. Finally, HCIA can generate an impact set at different granularity levels as a set of pairs composed of source code unit name and its probability of being impacted ($\langle n, p \rangle$).

There are several ways to compute the probability of a source code unit being impacted. For example, the association rule shown in Equation (1), proposed by Zimmermann et al.²², can be used to compute the probability.

$$P(A \rightarrow B) = \frac{N(A \cap B)}{N(A)} \quad (1)$$

In Equation (1), A and B are source code units in an application. $N(A \cap B)$ represents the *support value*, which is the number of times A and B have been changed together in the source code repository of the application. $N(A)$ represents the *frequency* of A, which is the number of times A has been changed. $P(A \rightarrow B)$ denotes the probability that B will be changed if A is changed and is also called the *confidence value*.

Although Equation (1) is effective, it assumes that the support value term only includes intentional co-changes, that is, co-changes where A and B are modified together for the same reason. In many cases, this assumption may not hold. For example, A and B may be related to separate features that are included in the same release. For instance, assume that $N(A) = 1$, $N(B) = 10$, and $N(A \cap B) = 1$. If A is the location of the proposed change, $P(A \rightarrow B) = 1$ which means that the change in A impacts B and B also needs to be changed. However, it is possible that A and B have been changed together incidentally in the past since B has been changed many times. As a result, changing A might not necessitate changing B. To avoid such bias, Equation (1) can be modified to require a minimum support value (i.e., only co-changes that occur a sufficient number of times are considered) as shown in Equation (2)²³.

$$P(A \rightarrow B) = \begin{cases} \frac{N(A \cap B)}{N(A)} & , N(A \cap B) \geq \text{MinSupValue} \\ 0 & , \text{otherwise} \end{cases} \quad (2)$$

A second method for calculating probabilities, proposed by Jashki et al.²⁴, is to use the Jaccard similarity coefficient²⁵. This method is shown in Equation (3).

$$P(A \rightarrow B) = \frac{N(A \cap B)}{N(A) + N(B) - N(A \cap B)} \quad (3)$$

In Equation (3), $N(A)$ and $N(B)$ again represent the number of times that these source code units have been changed in the source code repository of the application. The support value is divided by the frequency of A plus the frequency of B minus the support value. This equation considers the closeness of the methods to calculate the probability.

In our prototype implementation, developers are easily able to select any of the three equations or define any minimum support value for the probability calculation.

4.3 | Identifying the Callees of the Source Code Units

Almost all Android applications are Java-based. Thus, we use DependencyFinder¹⁴ in our prototype to identify the callees of the Android application source code units. The callee relationships are used to specify whether a source code unit directly calls any of the eGreedy methods. The use of eGreedy methods necessitates energy testing for the source code unit.

DependencyFinder analyzes compiled Java codes and builds dependency graphs. In the dependency graph, a dependency indicates that one element requires the presence of another element to function. DependencyFinder has 3 different element types: package, class, and feature.

TABLE 1 Dependencies.

Elements	Package	Class	Feature
Package	✓	✓	✓
Class	✓	✓	✓
Feature	✓	✓	✓

Table 1 shows dependencies that DependencyFinder can infer with a checkmark (✓). As the table shows, there are 9 dependency options including "package to package", "package to class", "package to feature", etc. We selected the "feature to feature" option of DependencyFinder

to obtain dependencies at different granularity levels. With this option, it is possible to identify the callees of the application packages, files, and methods by parsing the dependency graph.

5 | EXPLORATORY STUDY

We designed three case studies in which we use the prototype of ERU on ten Android applications to investigate the following research questions:

RQ1 – Applicability. Can ERU help to plan the application evolution process?

RQ2 – Energy Testing. Do proposed single source code unit changes ever result in not requiring energy testing for real applications using energy-greedy API methods?

RQ3 – Impact. How does the choice of history granularity, threshold (α) and equation for computing confidence values affect the energy testing requirement of proposed code changes?

The remainder of this section provides information about the selected applications and explains the experimental procedure that we followed.

5.1 | Procedure

We employed the prototype of ERU to determine whether energy testing is required (energy testing requirement) for the proposed changes in ten Android applications. We conducted three case studies and followed the same procedure for each study. The first step was to select the source code unit granularity of the study and to identify the proposed changes in the latest version of each application. Because we are interested in source code changes, all source code units (in “./java/” directory) at the selected granularity level were considered as possible proposed changes. Note that, these proposed changes do not include any non-code changes (e.g., changes in user interface (UI), Android SDK version, non-Java files, etc.). Then, the energy testing requirement of each proposed change were investigated using each of the equations described in Section 4.2 with different thresholds, 0.5, 0.6, 0.7, 0.8, and 0.9. The details of the case studies are presented below:

Case Study 1 - Method Level with Release History: The first case study was conducted at method level, and each application method was considered as a proposed change. The impact sets of the proposed changes were generated based on the release history of the application.

Case Study 2 - File Level with Release History: The second case study was conducted at the file level, and each source file was considered as a proposed change. The impact sets of the proposed changes were generated based on the release history of the application.

Case Study 3 - File Level with Commit History: Similar to the second study, the third case study was conducted at the file level. The difference is that the impact sets of the proposed changes were generated based on the commit history of the application.

Two different application histories from the application’s source code repository were used in these case studies. While release history consists of distributed versions of an application, commit history includes all the submitted commits during the evolution process of an application after its first distributed version. Basically, we obtained the impact sets of the proposed changes from source code changes between releases in the first and second case studies, and between commits in the third case study. Note that, we only considered the commits that contain source code changes that occurred in the current source code directory structure. Commits with only non-code changes or within a different directory structure (e.g., before migrating Eclipse to Android Studio) were ignored.

We have not conducted a case study at the method level with commit history. The only way we could find to obtain modified application methods in a commit is to use SVN repository although all of our considered applications are in Git repository. The underlying reason is the difficulty of migrating application’s Git repository to SVN repository due to many branching-outs and merges in Git that SVN cannot handle. Therefore, we manually created SVN repository for each application by committing only source code of the application in its releases with respect to release order for the first case study. To use same release history in the second case study, we automatically converted application SVN repositories to Git repositories. Lastly, we used existing application Git repositories in the third case study.

5.2 | Applications

Table 2 lists the applications that we used in our exploratory study. The first two columns, *Application* and *Description*, list the name of each application and a brief description of its functionality, respectively. The third and fourth columns, *# Releases* and *# Commits*, show the number of releases and commits that we used to create release history and commit history of each application from the application’s source code repository,

respectively. The fifth column, *Version*, indicates the latest version number of each application that was analyzed in our exploratory study. The next two columns, *# Files* and *# Methods*, provide the number of source files and the number of methods in the latest version of each application, respectively. To gather all method and file names, and the number of methods and files in the application, we used *DependencyFinder*. The number of lines of code, *LoC*, is reported in the final column and only counts lines of code in the source files of the application. For example, TintBrowser is a web browser application and it has 10 releases and 171 commits in its source code repository. Version 1.8 of TintBrowser consists of 96 Java files, 919 methods, and 19,647 lines of code.

TABLE 2 Considered applications.

Application	Description	# Releases	# Commits	Version	# Files	# Methods	LoC
AdAway	Ad bloker	19	48	3.1.2	55	317	9,421
AnkiDroid	A flashcard-based study aid	42	862	2.7	250	3,027	69,206
Budget	Financial tracker	11	84	4.2	38	305	5,564
Clover	4chan image board browser	22	663	2.2.0	230	2,117	37,612
Photo Manager	Search photos in local media	14	281	5.3.16	72	706	15,200
Red Moon	Screen filter for night time use	21	143	2.9.1	39	209	5,899
TintBrowser	Web browser	10	171	1.8	96	919	19,647
Tomdroid	Note taking application	11	574	0.7.5	50	446	12,123
Vanilla Music	Music player	38	1,184	1.0.45	85	1,000	25,036
Wikipedia	Official Wikipedia application	35	1,214	2.4.160	519	4,147	52,189

We chose these specific applications because they have available source code repositories and they are representative of common Android application types. These applications also vary in the number of releases, commits, files, methods, and lines of code.

5.3 | Cost

The amount of time needed by the prototype of ERU to provide energy testing feedback of the proposed changes depends on several factors. The main factors are the number of considered source code units as proposed changes, number of historical changes for a proposed change, and number of releases or commits in the application source code repository. Another factor is that obtaining methods name takes more time than file names due to parsing expenses. As there are several independent factors, it is difficult to make complexity analysis. In our case studies, while maximum time spent by the prototype was around 10 minutes (AnkiDroid and Wikipedia applications at method level), minimum time was around 20 seconds (Budget application at file level) for all proposed changes in an application. For real cases, the number of proposed changes can be significantly less than compared to our case studies. Therefore, the necessary time to get feedback regarding energy testing requirement of proposed code changes can decrease substantially. This time can be negligible compared to the time that would have been unnecessarily spent performing expensive energy testing.

6 | DATA ANALYSIS AND DISCUSSION

In this section we describe results of our study in terms of our research questions.

6.1 | RQ1: Applicability

Table 3 shows the data used to answer the first research question. We gathered data the ERU prototype produces and uses internally to determine energy tests to be executed/skipped for the proposed changes. The first column, *Application*, lists the name of each application. The remaining columns, *% Use eGreedy* and *% Known Impact*, show the percentage of proposed changes that use energy greedy API methods in eGreedy and the percentage of proposed changes whose impact is known, respectively. For example, in Tomdroid, 10.1 % of the proposed changes at the method level and 32.0 % of the proposed changes at the file level use energy greedy API methods. 20.9 %, 50.0 %, and 84.0 % of the proposed changes' impact are known at the method level with release history, at the file level with release history, and at the file level with commit history, respectively.

TABLE 3 Proposed changes.

Application	Method Level		File Level		
	% Use eGreedy	Release History	% Use eGreedy	Release History	Commit History
		% Known Impact		% Known Impact	% Known Impact
AdAway	19.2	15.1	40.0	87.3	83.6
AnkiDroid	5.5	29.2	24.8	64.8	63.6
Budget	14.8	20.3	57.9	60.5	92.1
Clover	4.3	13.4	22.2	48.3	74.8
Photo Manager	10.8	37.3	45.8	69.4	79.2
Red Moon	7.2	22.5	35.9	82.1	97.4
TintBrowser	11.1	13.4	43.8	60.4	87.5
Tomdroid	10.1	20.9	32.0	50.0	84.0
Vanilla Music	8.6	27.8	48.2	76.5	84.7
Wikipedia	3.2	23.4	16.0	66.5	75.0
Average	9.5	22.3	36.7	66.6	82.2

From the data shown in Table 3, we can observe that Android applications often use energy greedy API methods as all of the considered applications invoke some energy greedy API. However, the use of energy greedy API methods varies from application to application and ranges from 3.2 % (Wikipedia) to 19.2 % (AdAway) with an average value of 9.5 % for method level proposed changes. At the file level, it ranges from 16.0 % (Wikipedia) to 57.9 % (Budget) with an average of 36.7 %. As code changes may have an effect on the energy consumption of the applications since they contain energy greedy API methods, this data motivates the necessity of energy testing after the changes.

Developers currently anticipate performing energy tests after each source code change. However, a proposed change might not require all energy tests to be run, and time can be wasted on unnecessary energy tests. Therefore, providing feedback on executing/skipping energy tests for the proposed changes can help to improve the planning process of application evolution. ERU achieves this for the proposed changes whose impact is known.

Based on the data in Table 3, the percentage of proposed changes whose impact is known varies from application to application. For example, it ranges from 13.4 % (Clover and TintBrowser) to 37.3 % (Photo Manager) with an average value of 22.3 % for the proposed changes at the method level with release history. Another point is that selected source code unit granularity and history have influence on the known impact.

When we compare studies at method level and file level with release history, it is clear that the percentage of known impact increases significantly at the file level for the same application. This is an expected result since a file might consist of several methods and has a higher chance of being changed than a method. In addition, some of the application methods may not need to be changed once they have been defined (e.g., getter/setter methods). This may lower the percentage of known impact at the method level.

Studies at the file level with release history and commit history show that release history sometimes provides higher percentage of known impact than commit history (e.g., AdAway and AnkiDroid). This is possible when the source code directory structure has been changed in the existing application repository as we only examined the application source code changes in the current source code directory structure for commit history. For example, migrating Android applications built with Eclipse to Gradle causes directory structure changes. For release history, directory structure changes are not a constraint because release history that we created is independent from the directory structure since it only includes the source code of the application in its releases. Besides this exception, commit history provides a higher percentage of known impact than release history, as expected.

6.2 | RQ2: Energy Testing

One of the major criteria that developers take into account to plan their application evolution process is the total cost of testing. In response to code changes, both traditional and energy testing should be performed, especially for mobile applications. However, the high cost of energy testing significantly increases the total testing cost, which can adversely impact the planning process of application evolution. In our exploratory study, we investigated whether a proposed single source code unit change can ever result in not requiring energy testing.

Figures 5, 6 and 7 show the result of our investigation at method level with release history, file level with release history, and file level with commit history, respectively. These figures are faceted by the considered applications. In each facet, the y-axis shows the percentage of proposed

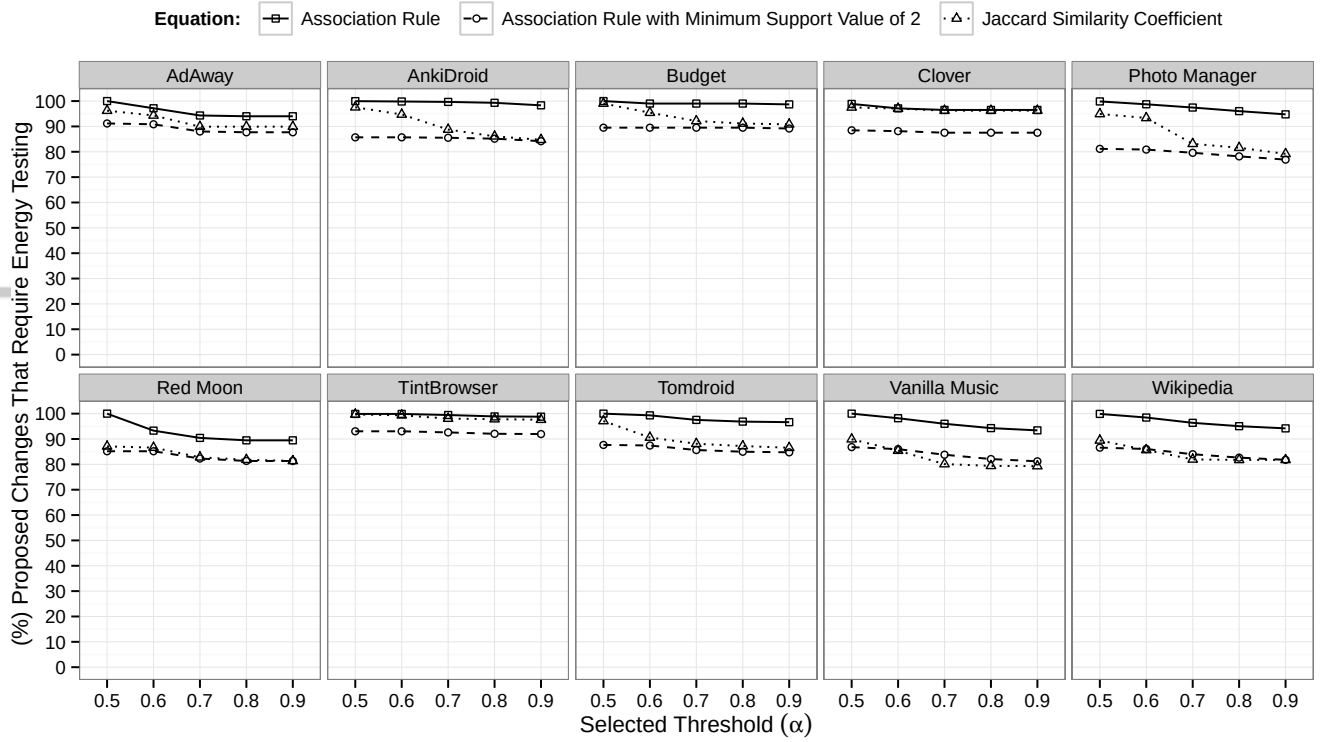


FIGURE 5 Energy Testing Feedback for Case Study 1: Method Level with Release History.

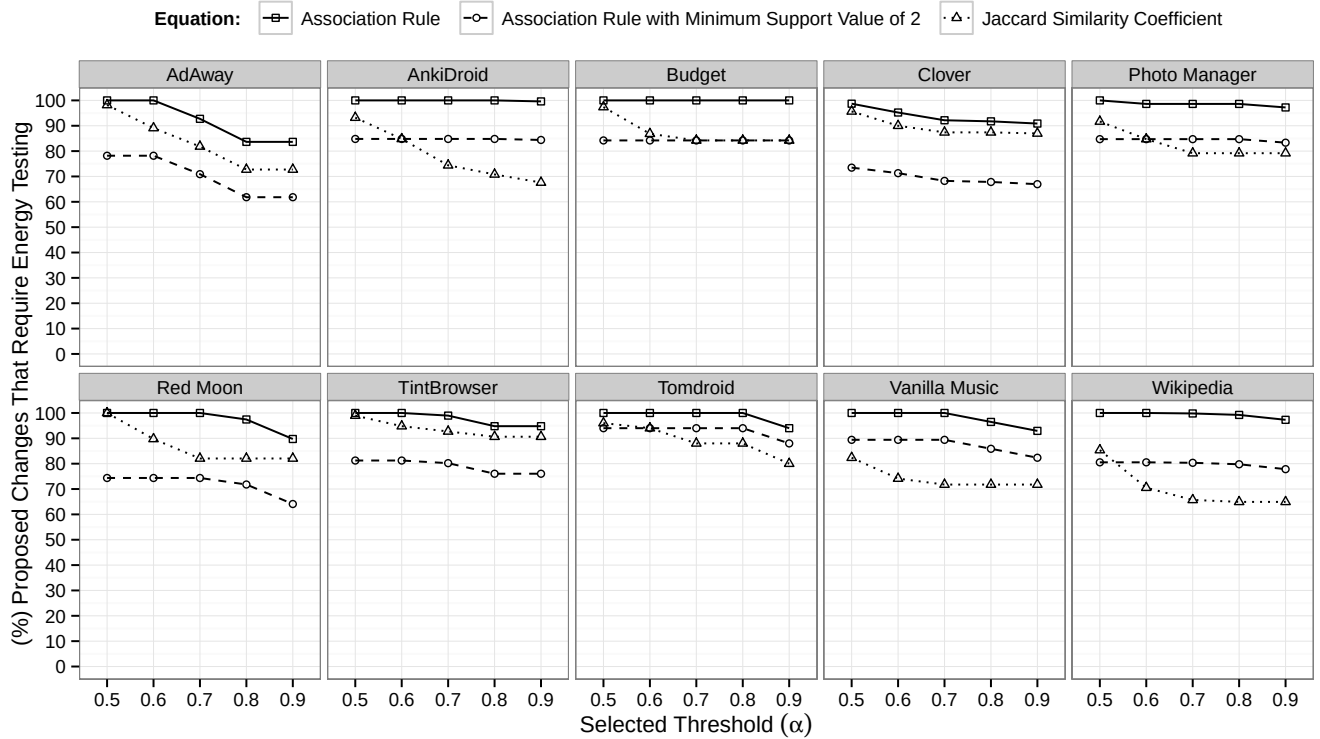


FIGURE 6 Energy Testing Feedback for Case Study 2: File Level with Release History.

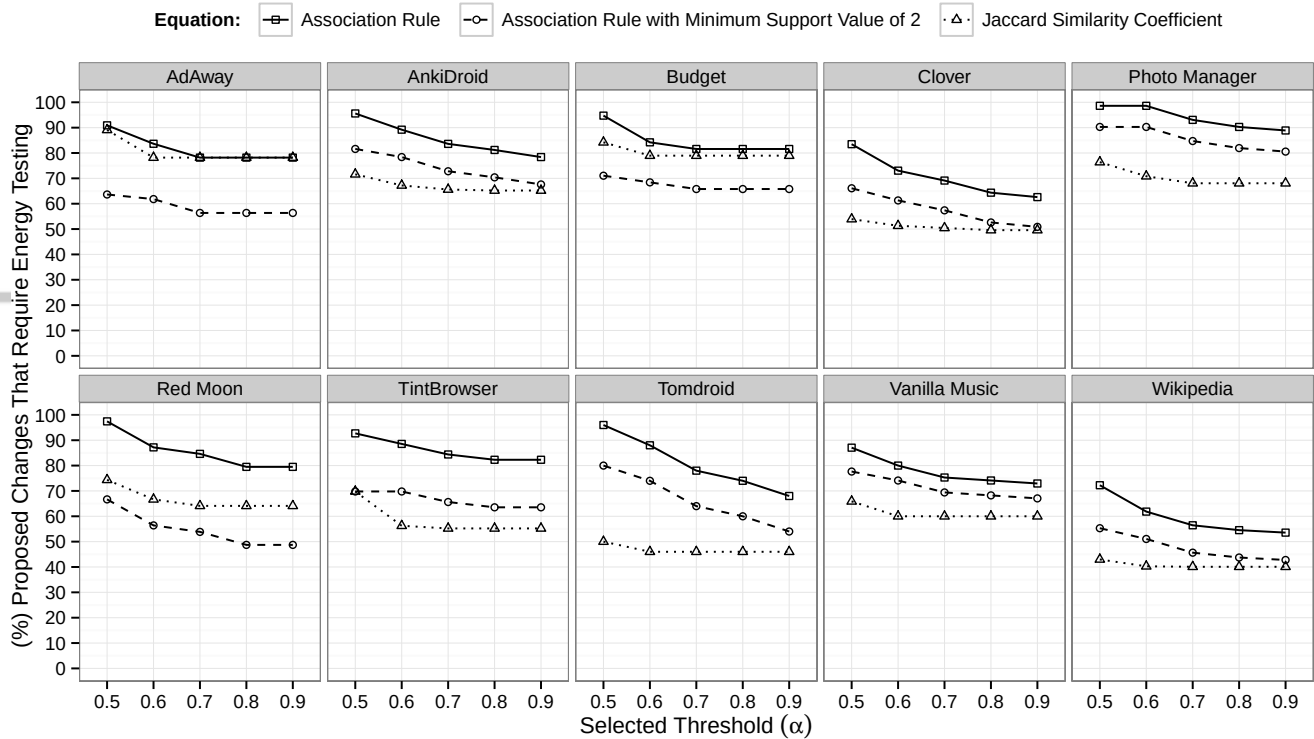


FIGURE 7 Energy Testing Feedback for Case Study 3: File Level with Commit History.

changes that require energy testing, and the x-axis shows the selected threshold. The plot lines indicate the equation that is used to calculate the probability of being changed.

As the figures show, in the majority of cases, energy testing is not always required for some of the proposed changes in an application. Although the result may vary in each study depending on selected threshold and equation, there is no case in which all of the proposed changes in an application require energy testing regardless of the selected threshold and equation except the Budget application for the association rule equation at the file level with release history. The underlying reason might be that the Budget application has limited release history and small number of source files.

6.3 | RQ3: Impact

We examined the data in Figures 5, 6 and 7 again to address the third research question.

History: Studies at the file level can be comparable in terms of history effect on the energy testing requirement of proposed code changes because the only difference between these studies is that the impact sets are generated based on release or commit history. When we compare the Figures 6 and 7, we can observe that using commit history most likely decreases the amount of required energy testing for the proposed changes. On average, the difference ranges from 9.0 % to 26.1 % in the threshold and equation, which is used to compute confidence values, configurations. The underlying reason is the possibility of over estimating the energy testing requirements due to coarse-grained release history. For example, modified application files between two releases are assumed to have been changed at the same time. However, it is possible that files have been changed and committed at different commits within two releases. While using fine-grained commit history may provide a more accurate change impact analysis result, using release history is still beneficial, as the result in Figure 6 shows that energy testing is not always required.

Threshold: Based on the data in Figures 5, 6 and 7, it is clear that the effect of the threshold varies according to application and equation used to compute confidence values. In general, we expected that selecting a higher threshold value would decrease the percentage of energy testing required for the proposed changes. However, it is not always the case. For example, the percentage of required energy testing for the proposed

changes in AdAway application is steady for the thresholds 0.7, 0.8, and 0.9 at the file level with commit history. Conversely, in some cases, selecting higher threshold significantly decreases the percentage of proposed changes that require energy testing. For example, increasing the threshold from 0.5 to 0.6 at the file level with commit history decreases the percentage of proposed changes that require energy testing 13.5 % for TintBrowser application.

Equation: Different equations can be used to calculate the probability of being impacted for source code units due to the proposed change in historical change impact analysis. From the data in Figures 5, 6 and 7, the plot lines are very similar for association rule and association rule with minimum support value of 2 in each study. The difference is that defining a minimum support value decreases the percentage of proposed changes that require energy testing. More interestingly, the Jaccard similarity coefficient equation is not consistent in and across case studies. It suggests more or less energy testing than association rule with minimum support value of 2 equation, although it most likely suggests less energy testing than the basic association equation.

6.4 | Threats to Validity

One of the most significant threats to the validity of our results is that we considered 131 API methods and all methods in 3 API packages as energy greedy. There might exist other energy greedy API methods. However, we believe that we added all known energy greedy API methods into our Energy Greedy APIs (eGreedy). eGreedy set is also expandable and new energy greedy API methods that might be revealed by future studies can be added to the set.

A more specific concern is the possibility of over estimating the required energy testing due to the coarse-grained release history of considered applications. While using application commit history eliminates this threat, it might not be applicable all the time. For example, we are not able to use commit history at the method level. Additionally, using release history is still beneficial since results of our case studies with release history show that some of the proposed changes do not require energy testing.

There are several additional threats to validity of our exploratory study. First, considered applications do not have energy tests and test coverage information publicly available. As a result, the prototype of ERU could only determine whether energy testing is required. Second, dynamic impact analysis can be more precise than static impact analysis. However, dynamic impact analysis does require actually applying changes and executing application source code to generate an impact set. Therefore, we chose to use a static impact analysis. Third, we only considered ten Android applications. Although we selected these applications to cover different application types, it is possible that they may not be representative of all applications. Lastly, some code changes may result in energy bugs that cause an increase in energy consumption of the application. Although energy bugs are most likely caused by invocation of energy greedy APIs especially for mobile applications, there is still a possibility that they might occur without using energy greedy APIs.

7 | RELATED WORK

Unlike our approach, which determines energy tests to be executed/skipped before making any code changes in a source code unit, the majority of existing work on energy testing has focused on minimizing the test suite with energy consideration^{3,5}, investigating the adequacy of test suite for energy testing⁴, finding energy bugs⁶, and reducing the energy consumption of test suite⁷.

For example, Jabbarvand et al.³ propose an energy-aware test suite minimization approach to test the energy properties of an Android application with the minimum set of tests. They used integer programming and a greedy algorithm to reduce the test suite size while maintaining the test suite coverage for energy-greedy segments of an application that consume more energy. Similar to our study, they consider energy greedy APIs to determine energy-greedy segments of an application¹⁵. Their result shows that they are able to minimize test suite size without losing effectiveness of the test suite in revealing most of the energy bugs such as wakelock bugs, recurring callback bugs, and loop bugs.

In a recent study, Jabbarvand and Malek⁴ developed an energy-aware mutation testing framework for Android applications to help developers evaluating the quality of their test suites for energy testing. They defined fifty energy-aware mutation operators in six categories which are location, connectivity, wakelock, display, recurring callback and loop, and sensor. Their framework takes an application source code, extracts the abstract syntax tree (AST), and applies to mutation operators in AST to create mutant versions of the application. After creating mutant versions, the test suite of application is run over the original and mutant versions while profiling the power consumption. Then, power profiles of the original and mutant versions are compared to observe whether mutation operators cause differences on the power trace of the test cases in test suite. Finally, the framework suggests that test cases which cause differences on the power trace are suitable for energy testing.

Li et al.⁵ optimize a test suite in terms of energy. They measure the energy consumption of the test cases using hardware and use this information to generate energy-efficient test suites by applying integer programming. While energy-efficient test suites that can be performed post-deployment testing on mobile and embedded systems have reduced energy consumption, their test coverage is equally effective with traditionally generated minimized test suites.

Banerjee et al.⁶ present an automated test generation framework. Their framework systematically generates tests to detect energy hotspots and bugs in Android applications by combining a graph-based search algorithm and guidance heuristics. After generating a test that is a user interaction scenario, the framework executes the test on a smartphone and measures energy consumption simultaneously using a power meter. Then, the framework examines energy bugs and hotspots in different categories such as hardware resources, sleep-state transition heuristics, background services, and defective functionality. While it detects an energy bug based on the statistical dissimilarities in energy consumption of the device before and after executing the respective application with the test, it determines an energy hotspot that causes anomalous energy consumption by using an anomaly detection technique.

Kan⁷ presents a technique to reduce the energy consumption of the CPU via the Dynamic Voltage and Frequency Scaling (DVFS) during the regression testing. This technique is conducted on the assumption that over the versions of a program that do not have significant changes in functionality, CPU-bound tests remain CPU-bound. Therefore, optimizing CPU frequency for execution of CPU-bound tests saves energy and helps to reduce the energy consumption of the test suite.

In addition to source code changes, another concern has been raised in Android application development with OLED display technology. Since energy consumption of OLED displays depends on the color displayed by each pixel, application UI color selections became an important aspect for optimizing energy consumption of the application. Therefore, researchers create display power models and develop automatic color transformation techniques to help developers making energy efficient applications through UI color selections (e.g.,^{26,27,28,29,30}).

An application's energy efficiency is one of the new criteria for selecting an application among applications that achieve the same purpose. To simplify the selection, Behrouz et al.³¹ have developed an approach to provide information about energy consumption of applications in the same category. Similar to this study, Saborido et al.³² also compared applications in the same category respect to their energy consumption.

Another area of related work is a recent group of studies that aim to provide energy-related information to developers by empirically investigating the underlying causes of energy usage. These studies include impacts of: performance tips³³, code obfuscation³⁴, code refactorings³⁵, design patterns³⁶, web servers³⁷, sorting algorithms³⁸, advertisements³⁹, API usage^{40,15}, and programming models⁴¹. In addition, researchers have investigated an application's energy consumption among versions⁴² and among separate implementations of the same application⁴³, the energy consumption impacts of selecting between different software systems that achieve the same purpose⁴⁴, how developers ask questions about energy usage⁴⁵, and how developers think about energy when they develop their software⁴⁶.

8 | CONCLUSION AND FUTURE WORK

Expensive energy testing can adversely impact the planning process of application evolution. In this paper, we have presented a new approach, ERU, that provides feedback on executing/skipping energy tests for proposed code changes to help developers plan their application evolution timeline effectively. ERU leverages change impact analysis and pre-computed API energy usage information. We employed the prototype of ERU to determine energy testing requirement of proposed changes in ten Android applications at different source code unit granularities with release or commit history.

The results of this study demonstrate that:

- (1) ERU can provide feedback on energy testing requirement of the proposed code changes.
- (2) Android applications most likely use energy greedy API methods.
- (3) Energy testing is not always required for a proposed single source code unit change.
- (4) The percentage from proposed changes that required energy testing varies application to application, and it is affected by selected history, threshold, and equation used to compute confidence values.
- (5) The prototype implementation of ERU provides initial evidence that ERU can be part of the application evolution cycle.

As such, we believe that getting feedback on executing/skipping energy tests for proposed code changes is positive news for application developers. By using the feedback information, developers can plan the application evolution process and make decisions on code changes more informatively.

Based on these conclusions, there are several potential areas for future work. First, we plan to develop a way to use commit history of the application at the method level. Second, we plan to investigate the best choice of the threshold and equation used to compute confidence values for ERU by analyzing real code changes and interviewing developers. Lastly, we plan to improve the prototype and prepare the first operational release.

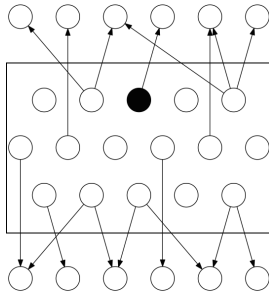
References

1. Wilke C., Richly S., GÄütz S., Piechnick C., ABmann U.. Energy Consumption and Efficiency in Mobile Applications: A User Feedback Study. In: Proceedings of the International Conference on Green Computing and Communications (GreenCom):134–141; 2013.
2. Apigee . Users Reveal Top Frustrations That Lead to Bad Mobile App Reviews. <http://apigee.com/about/pressrelease/apigee-survey-users-reveal-top-frustrations-lead-bad-mobile-app-reviews>; Accessed 1 October 2014; 2012.
3. Jabbarvand Reyhaneh, Sadeghi Alireza, Bagheri Hamid, Malek Sam. Energy-aware Test-suite Minimization for Android Apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis:425–436; 2016.
4. Jabbarvand Reyhaneh, Malek Sam. uDroid: An Energy-aware Mutation Testing Framework for Android. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering:208–219; 2017.
5. Li Ding, Sahin Cagri, Clause James, Halfond William G. J.. Energy-directed Test Suite Optimization. In: Proceedings of the 2Nd International Workshop on Green and Sustainable Software:62–69; 2013.
6. Banerjee Abhijeet, Chong Lee Kee, Chattopadhyay Sudipta, Roychoudhury Abhik. Detecting Energy Bugs and Hotspots in Mobile Apps. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering:588–598; 2014.
7. Kan E. Y. Y.. Energy Efficiency in Testing and Regression Testing A Comparison of DVFS Techniques. In: Proceedings of the 13th International Conference on Quality Software:280–283; 2013.
8. Pathak Abhinav, Hu Y. Charlie, Zhang Ming. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In: Proceedings of the 10th ACM Workshop on Hot Topics in Networks:5:1–5:6; 2011.
9. Hu Cuixiong, Neamtiu Iulian. Automating GUI Testing for Android Applications. In: Proceedings of the 6th International Workshop on Automation of Software Test:77–83; 2011.
10. Monkey . UI/Application Exerciser. <https://developer.android.com/studio/test/monkey.html>; Accessed 19 January 2016; 2016.
11. Gomez Lorenzo, Neamtiu Iulian, Azim Tanzirul, Millstein Todd. RERAN: Timing- and touch-sensitive record and replay for Android. In: Proceedings of the 2013 International Conference on Software Engineering:72–81; 2013.
12. Bohner Shawn A., Arnold Robert S.. *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press; 1996.
13. Android . Projects Overview. <https://developer.android.com/studio/projects/index.html>; Accessed 1 June 2017; 2017.
14. DependencyFinder. <http://depfind.sourceforge.net/>; Accessed 1 June 2016; 2016.
15. Linares-Vásquez Mario, Bavota Gabriele, Bernal-Cárdenas Carlos, Oliveto Rocco, Di Penta Massimiliano, Poshyvanyk Denys. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In: Proceedings of the 11th Working Conference on Mining Software Repositories:2–11; 2014.
16. Carroll Aaron, Heiser Gernot. An Analysis of Power Consumption in a Smartphone. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference:21–21; 2010.
17. Pathak Abhinav, Hu Y. Charlie, Zhang Ming. 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:29–42; 2012.
18. Pathak Abhinav, Jindal Abhilash, Hu Y. Charlie, Midkiff Samuel P.. What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services:267–280; 2012.

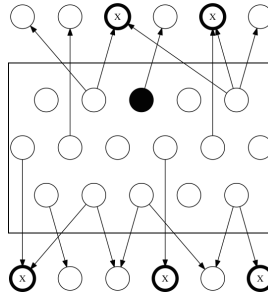
19. Zhang Jack, Musa Ayemi, Le Wei. A comparison of energy bugs for smartphone platforms. In: Proceedings of the 1st International Workshop on the Engineering of Mobile-Enabled Systems:25–30; 2013.
20. Lehnert Steffen. A review of software change impact analysis. https://www.db-thueringen.de/servlets/MCRFileNodeServlet/dbt_derivate_00024546/ilm1-2011200618.pdf; 2011.
21. Li Bixin, Sun Xiaobing, Leung Hareton, Zhang Sai. A survey of code-based change impact analysis techniques. *Journal of Software Testing, Verification and Reliability*. 2013;23(8):613–646.
22. Zimmermann Thomas, Weisgerber Peter, Diehl Stephan, Zeller Andreas. Mining Version Histories to Guide Software Changes. In: Proceedings of the 26th International Conference on Software Engineering:563–572; 2004.
23. Han Jiawei, Kamber Micheline, Pei Jian. *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2011.
24. Jashki Mohammad-Amin, Zafarani Reza, Bagheri Ebrahim. Towards a More Efficient Static Software Change Impact Analysis Method. In: Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering:84–90; 2008.
25. Jaccard Index. https://en.wikipedia.org/wiki/Jaccard_index; Accessed 30 October 2016; 2016.
26. Dong M., Zhong L.. Power Modeling and Optimization for OLED Displays. *IEEE Transactions on Mobile Computing*. 2012;11(9):1587–1599.
27. Wan M., Jin Y., Li D., Halfond W. G. J.. Detecting Display Energy Hotspots in Android Apps. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST):1–10; 2015.
28. Li Ding, Tran Angelica Huyen, Halfond William G. J.. Nyx: A Display Energy Optimizer for Mobile Web Apps. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering:958–961; 2015.
29. Linares-Vásquez Mario, Bavota Gabriele, Cárdenas Carlos Eduardo Bernal, Oliveto Rocco, Di Penta Massimiliano, Poshyvanyk Denys. 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:143–154; 2015.
30. Linares-Vásquez M., Bernal-Cárdenas C., Bavota G., Oliveto R., Penta M. Di, Poshyvanyk D.. GEMMA: Multi-objective Optimization of Energy Consumption of GUIs in Android Apps. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C):11–14; 2017.
31. Behrouz R. J., Sadeghi A., Garcia J., Malek S., Ammann P.. EcoDroid: An Approach for Energy-Based Ranking of Android Apps. In: 2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software:8–14; 2015.
32. Saborido R., Beltrame G., Khomh F., Alba E., Antoniol G.. Optimizing User Experience in Choosing Android Applications. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER):438–448; 2016.
33. Sahin Cagri, Pollock Lori, Clause James. From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. *Journal of Systems and Software*. 2016;117:307–316.
34. Sahin Cagri, Wan Mian, Tornquist Philip, et al. How does code obfuscation impact energy usage?. *Journal of Software: Evolution and Process*. 2016;28(7):565–588.
35. Sahin Cagri, Pollock Lori, Clause James. How Do Code Refactorings Affect Energy Usage?. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement:36:1–36:10; 2014.
36. Sahin Cagri, Cayci Furkan, Gutiérrez Irene Lizeth Manotas, et al. Initial explorations on design pattern energy usage. In: Proceedings of the First International Workshop on Green and Sustainable Software:55–61; 2012.
37. Manotas Irene, Sahin Cagri, Clause James, Pollock Lori, Winbladh Kristina. Investigating the Impacts of Web Servers on Web Application Energy Usage. In: Proceedings of the Second International Workshop on Green and Sustainable Software:16–23; 2013.
38. Bunse C., Hopfner H., Roychoudhury S., Mansour E.. Choosing the 'best' sorting algorithm for optimal energy consumption. In: Proceedings of the 4th International Conference on Software and Data Technologies:199–206; 2009.

39. Gui Jiaping, Mcilroy Stuart, Nagappan Meiyappan, Halfond William G. J.. Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers. In: Proceedings of the 37th International Conference on Software Engineering:100–110; 2015.
40. Manotas Irene, Pollock Lori, Clause James. SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In: Proceedings of the 2014 International Conference on Software Engineering:503–514; 2014.
41. Cohen Michael, Zhu Haitao Steve, Senem Emgin Ezgi, Liu Yu David. Energy types. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications:831–850; 2012.
42. Hindle A.. Green mining: A methodology of relating software change to power consumption. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories:78-87; 2012.
43. Arunagiri Sarala, Jordan Victor J., Teller Patricia J., et al. Stereo matching: Performance study of two global algorithms. *SPIE Proceedings*. 2011;8021:80211Z-80211Z-17.
44. Amsel N., Ibrahim Z., Malik A., Tomlinson B.. Toward sustainable software engineering: NIER track. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE):976–979; 2011.
45. Pinto Gustavo, Castor Fernando, Liu Yu David. Mining Questions About Software Energy Consumption. In: Proceedings of the 11th Working Conference on Mining Software Repositories:22–31; 2014.
46. Manotas Irene, Bird Christian, Zhang Rui, et al. An Empirical Study of Practitioners' Perspectives on Green Software Engineering. In: Proceedings of the 38th International Conference on Software Engineering:237–248; 2016.
47. Myers Glenford J., Sandler Corey, Badgett Tom. *The Art of Software Testing*. Wiley Publishing; 2011.



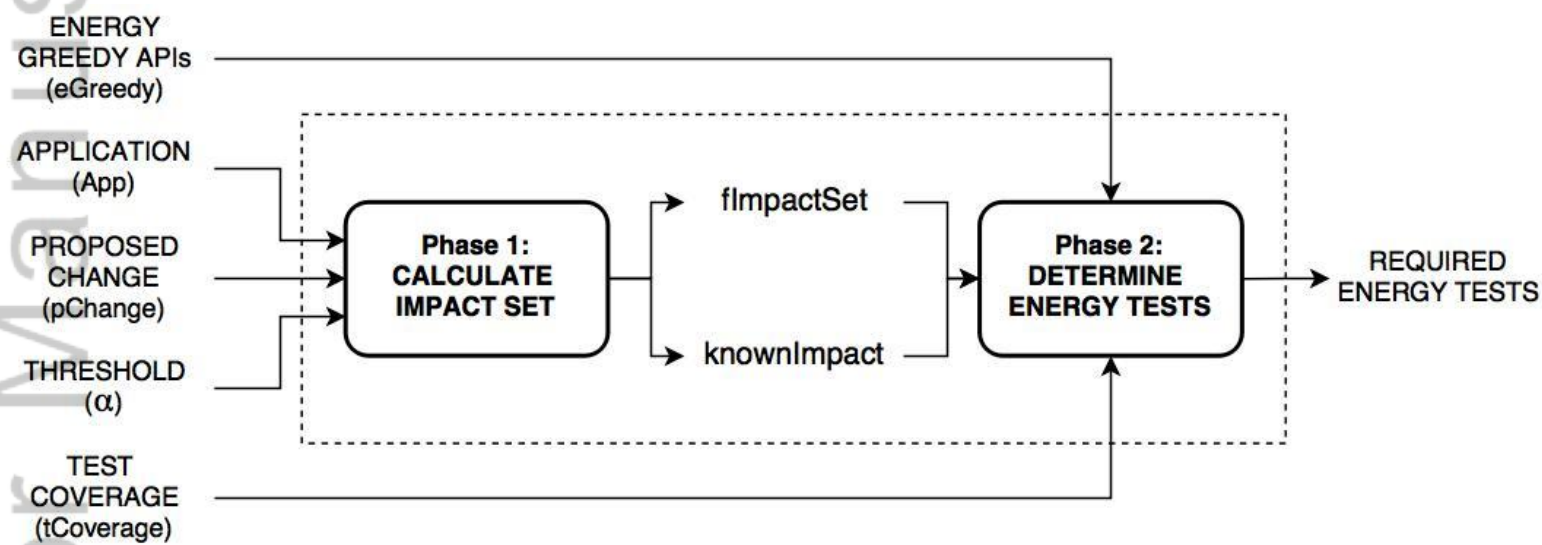


(a) Proposed change



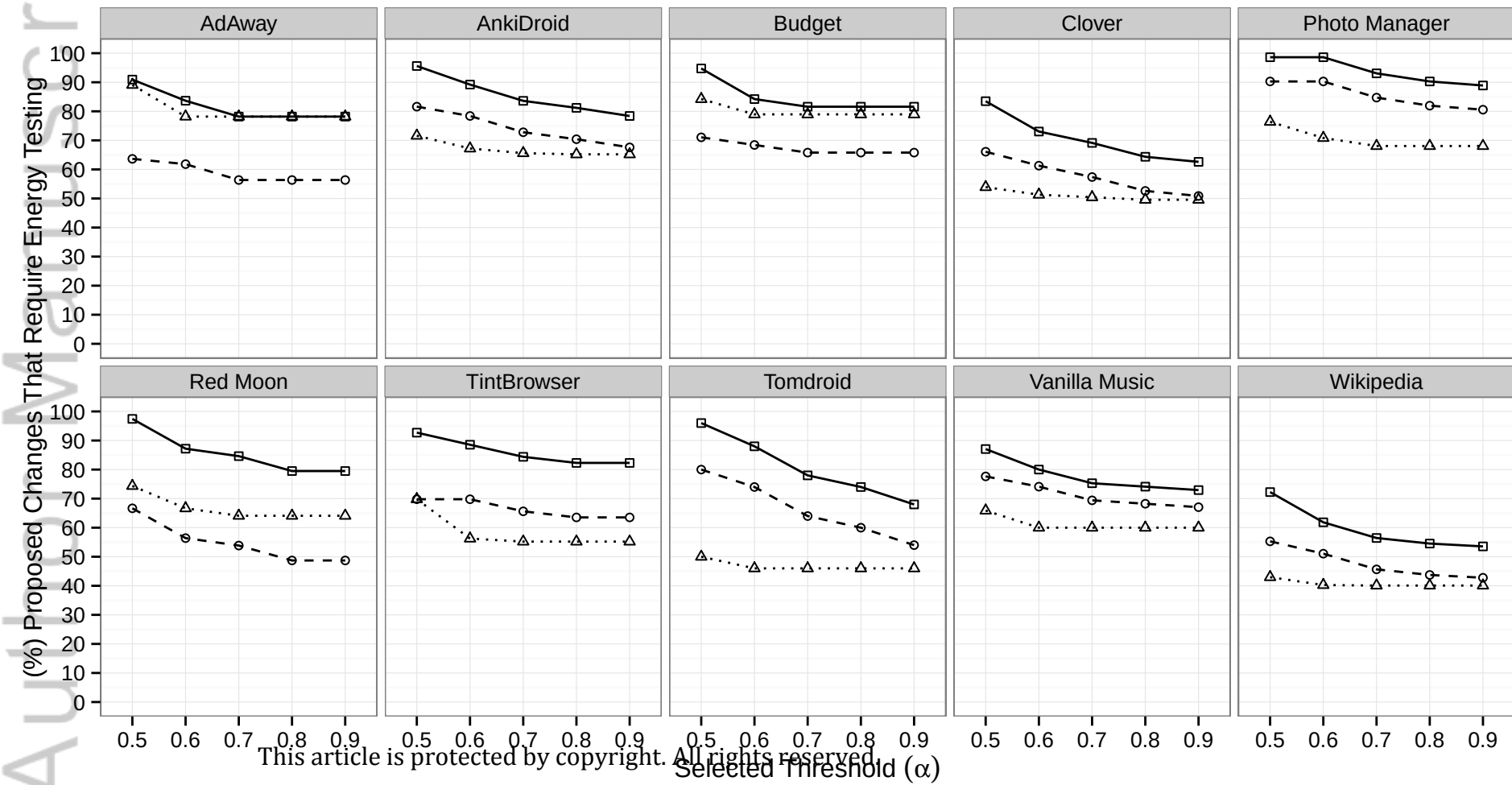
knownImpact = False

(b) After phases

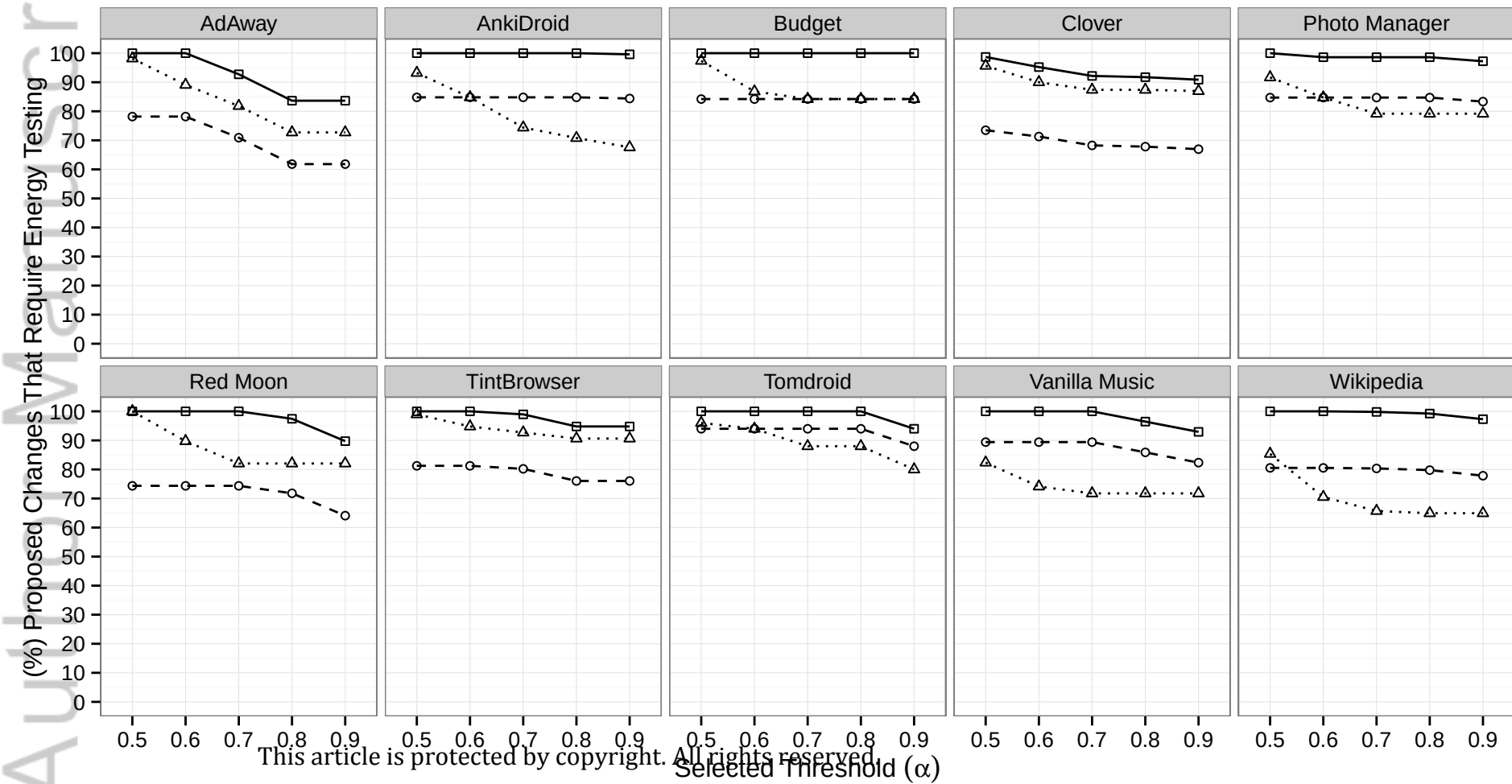


approach.jpg

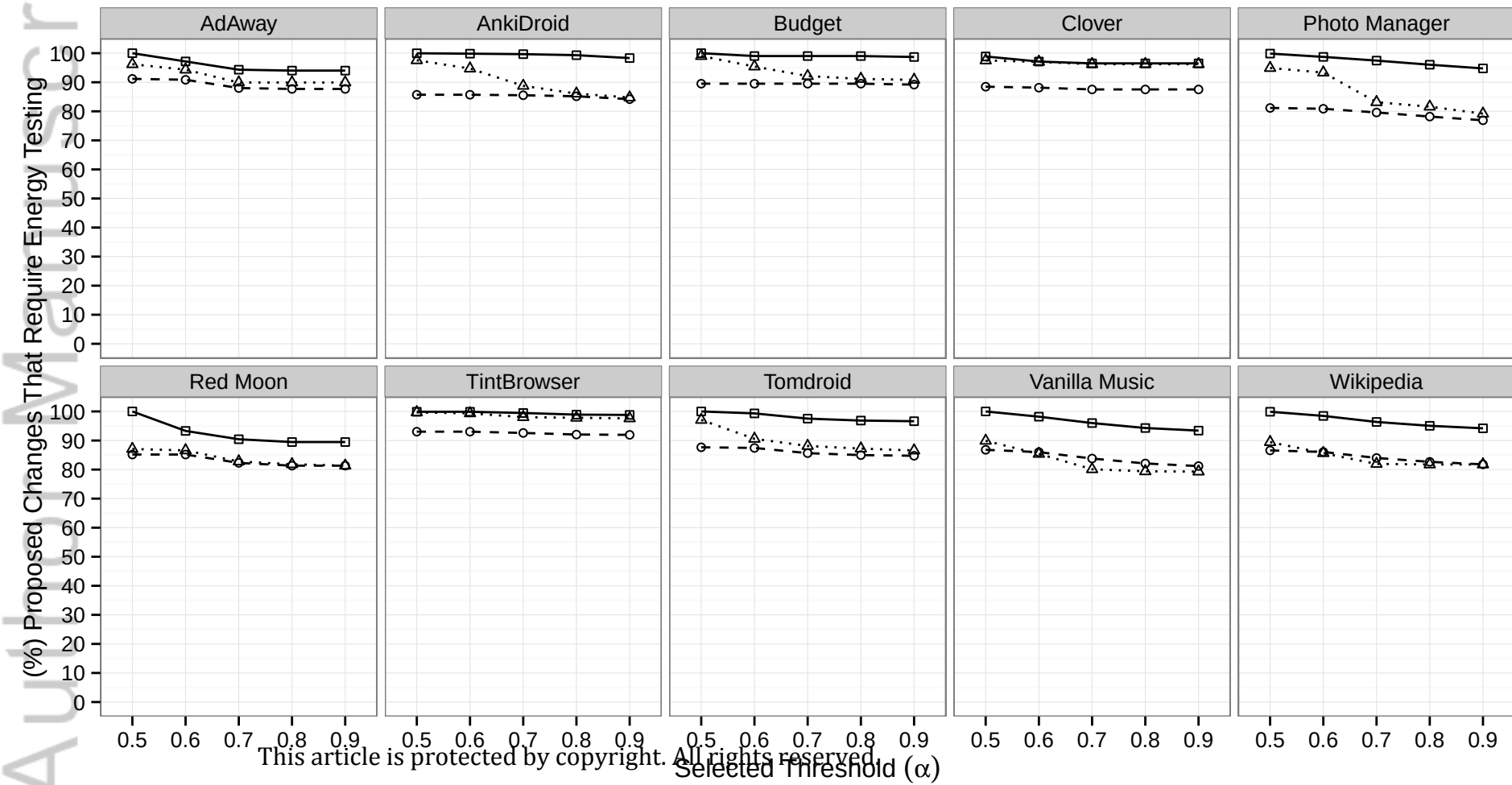
Equation:  Association Rule  Association Rule with Minimum Support Value of 2  Jaccard Similarity Coefficient

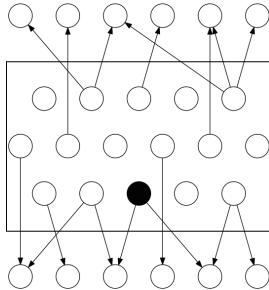


Equation:  Association Rule  Association Rule with Minimum Support Value of 2  Jaccard Similarity Coefficient

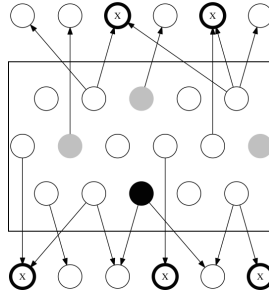


Equation:  Association Rule  Association Rule with Minimum Support Value of 2  Jaccard Similarity Coefficient



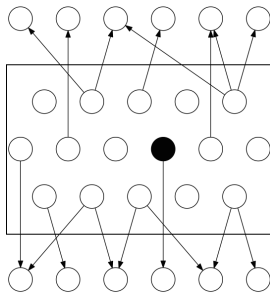


(a) Proposed change

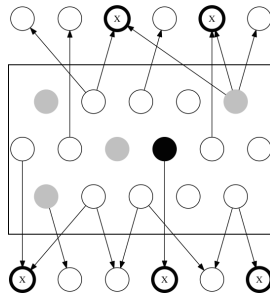


(b) After phases

knownImpact = True



(a) Proposed change



(b) After phases

knownImpact = True