2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software

EcoDroid: An Approach for Energy-Based Ranking of Android Apps

Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, Paul Ammann
Department of Computer Science
George Mason University
Fairfax, Virginia, USA
{rjabbarv, asadeghi, jgarci40, smalek, pammann}@gmu.edu

***Abstract*—The ever increasing complexity of mobile apps comes with a higher energy cost, creating an inconvenience for users on batter-constrained mobile devices. At the same time, due to the meteoric rise of the numbers apps provisioned on app repositories, there are often multiple apps from the same category (e.g., weather, dictionary) that provide similar features. In spite of similar functionality, the apps may present very different energy costs, due to the choices made in their design and construction. Given apps with similar features, users would prefer an app with the least energy cost. However, app repositories are currently lacking information about relative energy cost of apps in a given category, forcing the users to blindly choose an app for installation without a clear understanding of its energy implications. To address this issue, we have developed EcoDroid, an approach that ranks apps from the same category based on their energy consumption. To that end, EcoDroid uses both static and dynamic analyses to estimate energy consumption of apps in the same category and rank them accordingly. Our initial experiments have demonstrated the ability of EcoDroid in accurately ranking the energy cost of multiple apps from a particular category.**

I. INTRODUCTION

Android has become one of the dominant mobile platforms. Android app repositories, such as Google Play [3], have created a fundamental shift in the way software is delivered to consumers, with thousands of apps added and updated on a daily basis. Recent studies [11], [16] have shown energy consumption of apps to be a major concern for end users; however, app repositories provide no, or very limited, information as to the energy efficiency of apps provisioned on these repositories.

Given the proliferation of apps, it is often the case that many apps provide similar features, but with different implementation choices, thereby impacting their energy consumption. For instance, Google Play hosts dozens of highly rated *weather* apps, providing almost identical features, as depicted in Figure 1. These apps share highly similar features and ratings, but do not provide any readily available information as to their energy costs to help the user make an informed decision.

To make this information available for an entire app repository, a systematic approach is needed for repository maintainers or app developers to automatically produce *accurate rankings* of similar apps in terms of their energy consumption. Energy ranking of two apps α and β is *accurate* if α is ranked lower than β only when the actual energy cost of α is lower than β.

Accurate energy ranking of apps is challenging, as it requires estimating the representative usage of apps by users. Each app in a category must be exercised in a similar, uniform manner, to avoid gross over- or underestimation of an app's actual energy consumption. In addition, to accurately obtain energy rankings, an approach must account for all behaviors of an app that constitute significant energy consumption. Android API methods represent one significant source of energy consumption. These methods typically constitute 80% of an entire app's energy consumption [14]. Without exercising or representing these methods, the approach significantly underestimates the energy consumption of an app. Moreover, certain behaviors recur during an app's execution (e.g., due to loops, callbacks, scheduling mechanisms, etc.). Consequently, the repeated occurrence of such behaviors must be characterized appropriately.

To address these issues, we introduce EcoDroid, a novel approach for estimating and ranking the energy consumption of Android apps using a combination of dynamic and static analyses. EcoDroid uses automatically generated test-cases to execute apps and estimates their energy cost based on their API usage. These estimates take into account the energy cost of the paths executed by test-cases. Comparing energy consumption based on covered statements alone may introduce significant error due to the varying, and possibly low, coverage of generated tests using existing Android test automation tools. Moreover, the executed paths from generated test-cases may not cover energy-greedy behaviors of an app

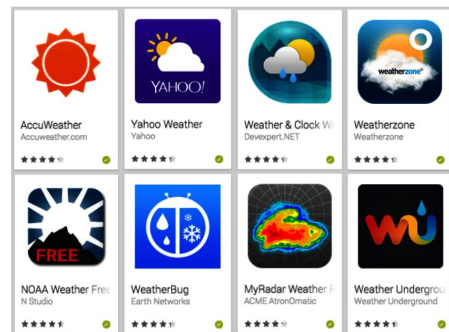

Fig. 1: A snapshot from Google Play showing suggested apps resulting from a "Weather" keyword search.

978-1-4673-7049-3/15 $31.00 © 2015 IEEE
DOI 10.1109/GREENS.2015.9

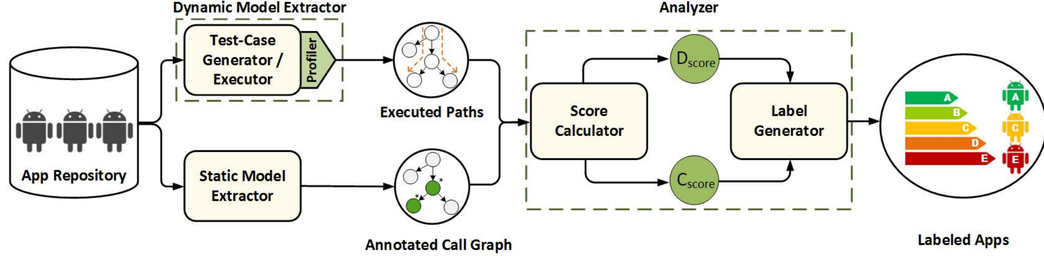

Fig. 2: EcoDroid Framework.

that have a substantial impact on its energy consumption. To address this limitation, EcoDroid leverages a novel static analysis to estimate the energy consumption of parts of an app that automatically generated tests do not execute. EcoDroid combines the static and dynamic results to produce an energy-consumption score for the app that is then used to rank the app. App repository maintainers would utilize EcoDroid's rankings to aid end users in deciding which apps in a category (e.g., weather or news reader) meets their energy-consumption needs. App developers would utilize EcoDroid energy score to evaluate and improve the efficiency of their apps

We demonstrate EcoDroid's accuracy through a pilot study on a set of apps from the *dictionary* category. Our study shows that the ranking produced by EcoDroid's combined use of static and dynamic analysis highly corresponds to the ground-truth ranking, which measures the actual energy consumption of apps. Furthermore, we demonstrate that EcoDroid's ranking is significantly closer to the ground-truth ranking than the ranking produced by an estimate based on dynamic analysis alone.

The remainder of this paper is organized as follows. Section II explains research challenges that arise when measuring and estimating the energy consumption of mobile apps. Section III describes the details of the approach. Section IV- V present the implementation and evaluation of EcoDroid. The paper then outlines related research and concludes with a discussion of future work.

II. Research Challenges

Measuring and estimating the typical energy consumption of an app entail determining its representative use-cases. Such use-cases can be provided by the app developers in the form of manually constructed tests, recorded from users' phone logs over a sufficient period of time, or generated using test automation tools. Unfortunately, very few apps in open-source repositories provide test-cases, which are mostly limited to unit tests and not representative of the app's typical usage. Similarly, collecting information about usage of apps from users is challenged by the privacy issues (e.g., logging of sensitive user data), as well as the overhead associated with collection of data.

Most prior research has studied energy behavior of mobile apps by manually utilizing and running apps several times [10], [13], [14]. However, such a manual process is neither systematic (e.g., may fail to exercise certain features of the app), nor scalable for use on an entire repository of apps. To address this issue, Li and colleagues [12] proposed leveraging *Monkey* [5], a widely used Android test generation tool, to automatically interact with apps and collect energy measurements. Monkey is a command-line tool that generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Li and colleagues considered *statement coverage* as a criterion to measure the extent to which automatically generated tests are representative of an app's typical use-cases. However, several issues arise with using statement coverage to represent typical usage of an app for analyzing its energy consumption.

Statement coverage, even when particularly high, can significantly misrepresent energy usage, because uncovered code may have a substantial impact on the energy cost. Prior research has shown that energy costs vary significantly across bytecodes [9], lines of code [13], and system APIs [14]. A test suite with high statement coverage may still not execute code that utilizes energy-greedy API calls. For example, APIs related to *GUI and image manipulation* tend to be particularly energy greedy. To measure and estimate energy consumption of an app, it is crucial to have tests that cover the energy-greedy parts of code.

Statement coverage for energy estimation is further unable to capture constructs involving statements that may execute many times. Specifically, statement coverage does not take into account statements that may execute in a loop or a recurring callback (e.g., due to thread scheduling or setting alarms on a mobile device). Mobile apps frequently utilize loops and call backs, making them important factors to consider for determining energy consumption of apps.

To overcome the limitations of statement coverage, we propose a new coverage criterion that indicates the degree to which energy-greedy statements of a program are tested. This new coverage criterion discriminates among different energy-greedy statements based on their energy cost and whether they re-execute due to recurring constructs, such as loops and callbacks.

Relying only on generated tests, even if they achieve high code coverage, can bias the energy measurement toward the executed statements. While dynamic analysis approaches, like testing and profiling, provide energy-related information about executed statements of an app, static analysis can be used in tandem to determine energy behavior of the unexplored statements. Prior research has leveraged program analysis to track energy-related information during execution and map

them to executed paths in order to measure/estimate energy consumption [10], [13]. To the best of our knowledge, no previous work has used static analysis to estimate the energy cost of the statements that are not covered by dynamic analysis.

EcoDroid overcomes the aforementioned challenges by combining static analysis with dynamic analysis in a complementary fashion to estimate the energy behavior of mobile apps. The reminder of the paper describes our proposed approach in detail.

III. Approach

The overall EcoDroid framework is shown in Figure 2. EcoDroid consists of three main components: (1) *Dynamic Model Extractor (DME)*, which automatically generates random tests and provides path information; (2) *Static Model Extractor (SME)*, which statically analyzes an app to obtain a call graph annotated with energy cost estimates; and (3) *Analyzer*, which combines information about executed paths and energy estimates from the annotated call graph in order to generate energy labels for each app.

We illustrate the manner in which EcoDroid computes the energy cost estimation of an app, using the call graphs of two hypothetical apps—*app1* and *app2*—as depicted in Figure 3. Each method in a call graph is annotated by a number representing the estimated energy consumption of the method, where greater values for a method indicate higher energy consumption.

A. Dynamic Model Extractor

DME is responsible for interacting with apps in order to generate test-cases and convert the test events (e.g., Android Intent messages and GUI events) to path information, which will be used later by Analyzer to estimate the energy consumption of the apps. DME accepts Android app package archives (APK) files as input and extracts the dynamic model of each app to reason about its energy-consumption behavior.

The dynamic model is defined as a set of paths, $P = \{p_1, p_2, ..., p_m\}$, where m is the number of test-cases generated during app execution. The path p_i is represented as a sequence of app-method and Android API invocations $\langle m_1\langle a_{1_1...1_k}\rangle, m_2\langle a_{2_1...2_k}\rangle, ..., m_n\langle a_{n_1...n_k}\rangle\rangle$, where m_i indicates invocation of the i_{th} method and $a_{i_1...i_k}$ denotes a sequence of Android APIs, a_{i_1} to a_{i_k}, called within method m_i. In the example shown in Figure 3, DME generates a set of two paths for each app. For example, the dynamic model for app1 is $P_1 = \{p_1, p_2\}$ and $p_1 = \langle A\langle a_{A_1}\rangle, C\langle a_{C_1}\rangle, D\langle a_{D_1}\rangle, G\langle a_{G_1}\rangle\rangle$. Each method presented in p_1 invokes one system API; however, our approach takes into account multiple APIs executed in a single method.

To extract a dynamic model, we utilize *Monkey* [5], an Android test-case generation tool, to run apps on Android devices. In each run, Monkey generates random sequences of user/system events, which correspond to executed paths in the dynamic model. To obtain dynamic path information, we have implemented a profiler module based on *Xposed* [6], which logs each method invoked by events generated by Monkey.

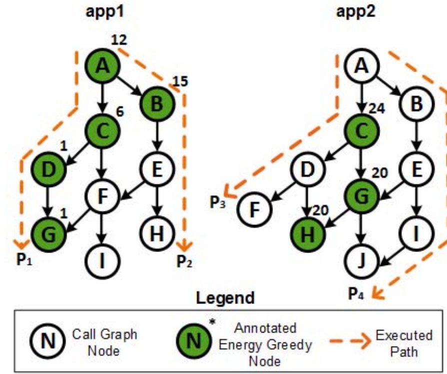

Fig. 3: Call graph and executed paths of two Android apps.

B. Static Model Extractor

To estimate energy consumption of an app, SME first extracts the app's *call graph (CG)*, which is a graph capturing the different possible invocation sequences within an app. We utilize *Soot* [15], a program analysis framework for Java, to statically analyze an Android app and extract its CG. EcoDroid annotates each CG node n (i.e., method in an Android app) with a *node score* s_n, which represents the estimated amount of energy consumed at each node. This score depends on the energy consumption of specific Android APIs invoked in n and the manner in which those APIs are invoked (e.g., whether an API is invoked in a loop). The precise energy consumption of individual APIs can be measured using hardware-based power monitors, such as Monsoon [4]. Linares-Vásquez et al. [14] have empirically studied and measured the energy consumption of Android APIs. Their results are used by EcoDroid to determine energy consumption of an Android API. Consequently, EcoDroid takes a list of such Android API energy measurements as input. SME calculates node score as follows:

$$s_n = r_n \times \sum_{i=0}^{m_n} \hat{e}_i \tag{1}$$

where r_n denotes the number of paths in the CG through which node n is reachable; m_n is the number of Android APIs used in the implementation of the method that node n represents; and $\hat{e}_i$ represents the energy consumption of an Android API i. We motivate and explain the two key components of s_n—r_n and $\hat{e}_i$—in the remainder of this section.

A naive assumption in energy cost estimation is to score each method n independently of the paths through which it may be invoked. This assumption is problematic because methods reachable along more paths in a CG are more likely to contribute in the energy cost of the app than methods reachable along fewer paths. To account for the energy effects of the number of paths through which a method can be invoked, s_n includes the component r_n. As shown in Figure 3, four paths can pass through node G in *app2*'s CG; thus, r_G = 4.

SME must discriminate CG nodes according to the APIs they invoke such that two nodes with distinct sets of API

invocations are likely to obtain different scores. This distinction is captured in the model: A node that invokes APIs with higher energy consumption has a greater value for $\sum_{i=0}^{m_n} \hat{e}_i$, compared to another node which calls less energy-greedy APIs. In addition, Representing recurring executions of APIs in a single statement is important for precise energy estimation. For example, APIs executed inside of loops are likely to contribute more to energy cost of an app than APIs executed outside of loops.

Two types of repeated executions are key to accurate energy estimation: (1) iterations over a data structure, and (2) services that continuously/periodically run, even if the app is idle. While Java loop structures are used to implement the former, Android provides services for scheduling repeated tasks for the latter (e.g. the AlarmManager, LocationManager, and ScheduledExecutorService). For example, methods of ScheduledExecutorService—which are used to create and schedule a recurring task—have parameters that define intervals between subsequent repeated executions. As shown in Figure 4, an instance of ScheduledExecutorService is created to execute a periodic action (i.e. receive updated weather data) every 30 seconds.

To include repeated executions in the model, SME considers an API i's energy consumption as follows:

$$\hat{e}_i = e_i \times f_i \times c \tag{2}$$

where e_i is the pre-measured energy consumption of API i given as an input; c denotes the number of times the API i is expected to execute in a loop; and f_i denotes the frequency for which i is expected to execute when scheduled as part of a repeated task. For c, SME extracts either (1) a constant loop bound, if such a bound can be obtained statically, or (2) if a loop bound cannot be determined statically, it assumes a configurable number of iterations. f_i is defined as $f_i = \frac{t_{tce}}{T}$, where t_{tce} is the average test-case execution time determined by DME, and T is the time period between executions of i.

To calculate f_i, EcoDroid first extracts the timing parameter of the corresponding APIs. To that end, we consult Android API documentation to specify the time-related parameters. For instance, consider `scheduleAtFixedRate(Runnable command, long Delay, long period, TimeUnit unit)`, the API method of ScheduledExecutorService in Figure 4. The third parameter, `period`, represents the duration between successive executions for the associated service and is used to determine the value of T. In Figure 4, t_{tce} is determined by DME to be 5 minutes and T is set to 30 seconds, resulting in f_i = 10. Consequently, the energy cost of all APIs called in the *updateWeather* method are multiplied by a factor of 10, as they could be invoked at most 10 times by ScheduledExecutorService during the 5-minute test-case execution.

C. Analyzer

As depicted in Figure 2, Analyzer's *Score Calculator* computes (1) the *dynamic cost*, D_{score}, from a set of executed paths obtained from DME, and (2) the *coverage score*, C_{score}, from the annotated CG produced by SME. Due to the limited coverage of state-of-the-practice automatic test-case generation tools for Android (e.g., Monkey), Analyzer normalizes D_{score} with a coverage score, C_{score}, to equitably compute estimated energy of apps for further comparison. For a set of paths P, *Score Calculator* computes the dynamic cost as follows:

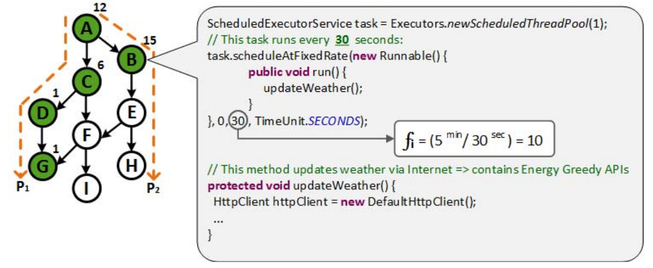

Fig. 4: Parts of the code corresponding to the call graph of app 1 shown in Figure 3.

$$D_{score} = \frac{\sum_{i=0}^{\rho} \sum_{j=0}^{\alpha_i} e_{ij}}{\rho} \tag{3}$$

where ρ denotes the number of paths; and α_i denotes the number of APIs invoked in path i. The e_{ij} parameter indicates the amount of energy consumed by the j_{th} API called in the i_{th} path.

Score Calculator also computes a hybrid coverage score, C_{score}, by combining static and dynamic models from SME and DME. To that end, *Score Calculator* maps the executed paths to the annotated CG and computes the C_{score} as follows:

$$C_{score} = \frac{\sum_{i=0}^{m} s_i}{\sum_{j=0}^{n} s_j} \tag{4}$$

where n is the total number of nodes in the annotated CG; and m is the number of nodes that are covered in at least one path.

C_{score} aims to compensate for the potentially low coverage of generated test-cases. To that end, the numerator considers CG nodes covered by paths, and the denominator represents all the nodes in the CG. As the number of nodes in the CG covered by executed paths grows, C_{score} increases (up to a value of 1), thereby executed paths more thoroughly cover the annotated CG, and D_{score} more accurately represents the typical energy consumption of the app under analysis.

Once *Score Calculator* produces D_{score} and C_{score}, *Label Generator* computes an overall energy estimate e_{index} as follows:

$$e_{index} = {}^{D_{score}}/_{C_{score}} \tag{5}$$

Based on this overall estimate, *Label Generator* assigns each app a label from A to E (recall Figure 2). Label A represents apps that are most energy efficient, while E designates those that are most energy expensive. To generate labels, *Label*

Generator ranks apps based on their e_{index}, categorizes them into 5 groups, and assigns labels to each group. By utilizing an ordinal scale, end users can easily distinguish and compare apps in terms of their energy consumption. Moreover, since our goal in this work is to determine the proper energy labels for similar apps, rather than determining the exact energy cost of apps, our approach is resilient to inevitable small estimation errors.

EcoDroid avoids a bias in analysis caused by execution of paths that consume an extremely low or high amount of energy. This is achieved through the normalization of D_{score} by C_{score}. Specifically, for executed paths with particularly high energy consumption—and thus a high D_{score}—in the annotated CG, the value of C_{score} is also high. This results in an e_{index} that is not severely affected by the particularly high energy consumption of the executed paths. A similar phenomenon occurs for executed paths with particularly low energy consumption.

To illustrate, consider the example in Figure 3. Taking only the energy cost of executed paths into account, *app1* purportedly consumes more energy than *app2*. That is, executed paths in *app1* hit more energy-greedy nodes, nodes in colored green, compared to *app2*. However, this conclusion is inaccurate since the CG of *app2* contains energy-greedy nodes that although are not covered by current paths, but may highly contribute to actual energy cost of the app. Thus, to reduce the bias of purely dynamic estimation on energy cost, D_{score}, we normalize it with C_{score}, to equitably compute estimated energy cost of an app.

IV. Implementation Prototype

This section describes some of the key implementation choices underlying EcoDroid's prototype. The DME component takes the APK file of an app as input and uses Monkey to interact with apps and generate random test-cases. While running Monkey on a mobile device, a rooted Nexus 5 with a Qualcomm Snapdragon chipset, a module implemented using the Xposed [6] framework records the invocation of methods and system APIs in a log file. The log file is later processed to extract information about the executed paths in each app. Xposed instruments the root Android process (called *Zygote*), rather than instrumenting an app's implementation. Thus, EcoDroid does not modify an app's APK file. The major advantages of using run-time process instrumentation over modifying individual apps are scalability and framework generalization. An Xposed module pinpoints methods and system APIs for any app installed on an Android device, precluding the need to instrument and modify every single app under study. Additionally, instrumentation of APK files changes the signature of apps, which might prevent their proper execution.

SME uses *Dexpler* [7] to translate an app's Android Dalvik bytecode into Soot's intermediate representation language. The Android platform is event-driven and leverages *implicit invocations*, i.e., method invocations performed by the Android platform in response to an event. Soot does not extract implicit invocations. To support such invocations, SME extends the default call-graph generator of Soot so that the resulting call graph includes them. To generate a call graph that takes implicit invocation into account, we need to include callbacks of an app. These are Android APIs that no other part of the app explicitly invokes. To that end, we traverse the nodes of the corresponding control-flow graph in a depth-first manner, and connect all nodes that make implicit invocations with the corresponding callback nodes. SME then traverses the generated call graph to calculate node score s_n for each node of the call graph and annotates the nodes with these scores.

TABLE I: Ranking of subject apps based on energy cost measured/estimated by Trepn, EcoDroid, and dynamic approach.

Apps	Trepn Rank	EcoDroid Rank	Dynamic Rank
com.amaltus.rt	1	1	3
an.FilipTranslate	2	2	2
com.cnasoft.dictek	3	3	6
com.appdjinnis.android.thesaurus	4	5	1
org.smartdict	5	4	5
com.merriamwebster	6	6	4

V. Evaluation

We have conducted a preliminary evaluation of EcoDroid to assess its overall accuracy in ranking apps from a given category according to their energy costs. This section first describes our experiment setup, followed by the results.

Apps are categorized under 26 classes on Google Play. However, we divide each of the pre-defined categories in Google Play into sub-categories, such that apps in the new categories have similar set of features. To categorize apps, we manually defined sub-categories (e.g., *Dictionary*, *Calendar*, and *Calculator*). We then used a modified version of Google Play Crawler [2] to start from one app in the defined sub-categories and find its closely *related apps*. Crawling for each subcategory continues until a pre-specified number of apps are found or no new apps are added.

For our preliminary experiment described in this paper, we chose 6 apps from the *Dictionary* category, which are among the top dictionary apps on Google Play. The reason behind selecting dictionary apps is that they have simple and limited use-case scenarios that can be explored and executed in a reasonable time, allowing us to obtain accurate ground-truth estimates.

To obtain the ground-truth estimates, we defined use-cases and ran each app several times with different settings. Example use-cases include looking up, adding, and deleting words in a dictionary. Apps from other categories may require specialized domain knowledge to identify typical use-case scenarios. For future work, we intend to conduct a comprehensive user study to reliably and accurately obtain typical usage for apps from such categories. We then used Trepn [8], a profiler tool that measures actual energy consumption of Android apps, to obtain ground-truth energy costs of apps. Trepn is designed to measure an app's effect on power, data, and CPU. It helps developers to observe how programming choices affect

power consumption in order to write power-aware apps. Trepn uses a series of sensors embedded on Qualcomm Snapdragon chipsets to monitor power consumption.

Since the majority of prior energy estimation approaches are based on dynamic analysis techniques, and one of the key contributions of EcoDroid is its hybrid static and dynamic analysis approach to energy estimation, we also compared EcoDroid against a purely dynamic solution. The *Dynamic* solution we used in our experiments computes the D_{score} to estimate the energy consumption of an app (recall Section III)

Table I shows the ranking of subject apps according to energy values computed by the three approaches: Trepn (ground-truth), EcoDroid, and the dynamic approach. The similarity of the rankings are measured using Spearman's rank correlation coefficient, which can take values from -1 to $+1$. A coefficient of $+1$ indicates a perfect association of ranks; a coefficient of 0 indicates no association between ranks; and a coefficient of -1 represents perfect negative association of ranks. We calculated Spearman's coefficient between Trepn and EcoDroid, as well as Trepn and the dynamic approach. The coefficient of Trepn's and EcoDroid's rankings is 0.943; the coefficient of Trepn's and the dynamic approach's rankings is 0.371. Thus, EcoDroid's ranking very closely resemble the ground-truth ranking, while the dynamic approach's ranking is nowhere close to the ground-truth ranking.

Figure 5 visualizes the *relative energy ranks* produced by the three approaches for the six studied apps. A relative energy rank is defined as $r_{app} = \frac{score_{app}}{maxScore}$, where $score_{app}$ is the energy score for an app computed by one of the approaches and $maxScore$ is the largest score obtained among apps that are to be ranked. r_{app} normalizes the energy cost of each app by the energy cost of the app with the greatest energy score. Consequently, for each of the three approaches, each app obtains a score relative to each other. As shown in Figure 5, the ranks of 6 studied apps provided by EcoDroid are very similar to that obtained by Trepn. On the contrary, the ranks produced by the purely Dynamic solution are very different from that of Trepn.

Our experiments, thus, corroborate the benefits of a hybrid static and dynamic analysis approach for energy ranking, as EcoDroid is able to produce a more accurate ranking compared to the purely dynamic approach that has been widely used in profiling the energy cost of mobile apps.

VI. Related Work

There is a large body of work on energy consumption of Android apps. Prior related studies can be categorized in two ways: 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 [10], [13]. These software-based approaches build models and capture model parameters from programs using static-analysis techniques. Compared to our approach, most of these techniques do not utilize power-measurement devices, or do not consider the hardware platform. These approaches

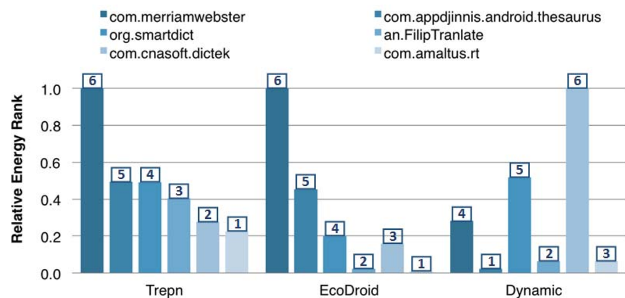

Fig. 5: Visualization of rankings provided by Trepn, EcoDroid, and dynamic approach.

over-approximate energy estimation and cannot be used for a fair comparison among different apps.

Studies in power measurement make use of specialized hardware, such as Monsoon, and map the sampled measurements to execution traces to determine an app's energy consumption at various granularities. Our approach leverages the power measurements of Linares-Vásquez et al. [14] to obtain energy-consumption estimates for Android APIs.

To the best of our knowledge, EcoDroid is the first work that has attempted to rank Android apps according to their energy consumption using a hybrid static and dynamic analysis approach. Closely related to our approach are vLens [13], eLens [10], and the work by Li et al. [12]. vLens provides fine-grained estimates of energy consumption at the code level by combining program analysis and per-instruction energy modeling. vLens can be used by developers for estimating the energy consumption of their apps. Unlike EcoDroid, vLens assumes the input workload of the approach is provided by developers who are aware of an app's behavior. Although the test-cases provided by developers is the best way to interact with apps, many apps from app repositories do not come with test-cases. In addition, for the purpose of energy estimation, the test-cases provided by developers might be biased, and may not represent the typical usage of apps.

eLens is an extension of vLens, which is able to calculate source line-level energy consumption information. eLens combines hardware-based power measurements with program analysis and statistical modeling. Similar to vLens, eLens assumes that the test-cases are provided by developers. eLens and vLens are orthogonal to EcoDroid, as they calculate energy consumption of mobile apps at different levels of granularity. EcoDroid uses the pre-measured energy consumption of Android APIs as an input to the approach. This list of APIs and measured values can be replaced by the measurements obtained from other similar approaches (e.g., eLens and vLens).

Li et al. [12] conducted an empirical study to discover quantitative and objective information about the energy behavior of apps that can be used by developers. To interact with an app, the authors utilize Monkey for test generation and vLens to estimate the energy cost of an app. Although they excluded test-cases with less than 50% statement coverage, our preliminary evaluation demonstrates that statement coverage suffers from limitations as a criterion for test-case quality

measurement in the context of app energy consumption. In addition, EcoDroid estimates the energy cost by considering information from program statements that are not covered by generated tests. Our preliminary results show that EcoDroid's approach for estimating energy cost is more accurate compared to relying only on executed paths.

VII. Limitations of EcoDroid

Our approach aims to generate an accurate energy ranking for Android apps. As a result, it does not intend to estimate the exact energy consumption of apps. Therefore, estimates produced by EcoDroid can tolerate a certain level of inaccuracy as long as the ranking remains the same. We have attempted to address the major sources of inaccuracy. However, we also had to strike a balance by making certain assumptions to improve EcoDroid's scalability and ease of applicability, which may impact the accuracy of its estimates.

EcoDroid is a lightweight approach that helps repository maintainers and developers to understand energy behavior of apps from the same category by assigning energy labels to them. It does not require developers to use specialized hardware, or instrument the apps, which may cause run-time failure. EcoDroid takes the pre-measured energy consumption of APIs (or lines of code) as an input and estimates the energy cost according to these values. Therefore, the accuracy of estimation depends on the precision of these pre-measured values. We use the outcome of a published study [14] that measures the energy consumption of APIs observed in diverse apps from Google Play.

In order to calculate s_n in Equation 1, we assumed that all the APIs used in the implementation of node n would be reachable. However, it is possible that APIs are invoked through branches, where only a portion of them are invoked through node n. One naive assumption is to consider the same probability for each branch to be taken and compute s_n as $s_n = r_n \times \sum_{i=0}^{m_n} \hat{e_i} \times p_i$, where p_i is the probability that e_i is invoked through the node. However, the probability of different branches are different and calculating precise probability is a challenge in its own right. Therefore, EcoDroid simply assumes that all the APIs within the implementation of a method will be called through its invocation.

EcoDroid calculates $\hat{e_i}$ using Equation 2. If a loop bound can be obtained from the source code (recall Section III-D), EcoDroid leverages this value for estimating $\hat{e_i}$. Otherwise, it assumes a constant number of iterations for the loop, which can be configured by the user. The second method may impact the accuracy of the approach. However, from exploring several open-source apps from F-Droid [1], we discovered that loops in such apps mostly iterate over data structures where the bound can be obtained statically.

VIII. Conclusion and Future Work

Energy is a critical resource for mobile devices. Optimizing the energy efficiency of mobile apps can greatly increase user satisfaction. While users often have a choice of numerous apps providing similar functionality, app repositories currently do not provide users with the energy information that would allow them to make informed decisions.

To address this issue, we presented EcoDroid, a hybrid static and dynamic analysis technique that estimates the energy cost of apps from a given category and ranks them accordingly. Our preliminary evaluation of EcoDroid on six real-world apps shows that EcoDroid accurately ranks apps according to their energy consumption. Since our goal in this work is to determine the proper energy labels for similar apps, rather than determining the exact energy cost of apps, our approach is resilient to inevitable small estimation errors. App repository maintainers can utilize EcoDroid's rankings to aid end users in deciding which apps in a category (e.g., weather or news reader) meets their energy-consumption needs. App developers can utilize EcoDroid energy score to evaluate and improve the efficiency of their apps

IX. Acknowledgment

This work was supported in part by awards D11AP00282 from the US Defense Advanced Research Projects Agency, H98230-14-C-0140 from the US National Security Agency, HSHQDC-14-C-B0040 from the US Department of Homeland Security, and CCF-1252644 from the US National Science Foundation.

References

[1] "F-droid," https://f-droid.org.
[2] "Google play crawler," http://goo.gl/BFc51M.
[3] "Google play market," http://play.google.com/store/apps.
[4] "Monsoon," http://goo.gl/8G7Xgf.
[5] "UI/Application Excersizer Monkey," http://goo.gl/6EN2gi.
[6] "Xposed Framework," http://goo.gl/9UKa0Z.
[7] A. Bartel, J. Klein, Y. LeTraon, and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *The Intl. Workshop on State of the Art in Java Program analysis*, 2012.
[8] L. Ben-Zur, "Developer Tool Spotlight - Using Trepn Profiler for Power-Efficient Apps," http://goo.gl/ESxXzi.
[9] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," in *The Intl. Workshop on Green and Sustainable Software*, 2012, pp. 1–7.
[10] ——, "Estimating mobile application energy consumption using program analysis," in *The Intl. Conf. on Software Engineering*, 2013.
[11] M. V. Heikkinen, J. K. Nurminen, T. Smura, and H. Hämmäinen, "Energy efficiency of mobile handsets: Measuring user attitudes and behavior," *The Telematics and Informatics*, 2012.
[12] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of android applications," in *The Intl. Conf. on Software Maintenance and Evolution*, 2014.
[13] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *The Intl. Symposium on Software Testing and Analysis*, 2013, pp. 78–89.
[14] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *The Working Conf. on Mining Software Repositories*, 2014.
[15] R. Valle é-Rai, P. Co, E. Gagnon, L. Hendren, and V. Lam, P.and Sundaresan, "Soot-a java bytecode optimization framework," in *The Conf. of the Centre for Advanced Studies on Collaborative research*, 1999.
[16] C. Wilke, S. Richly, S. Gotz, C. Piechnick, and U. Aßmann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *The Internation Conf. on Green Computing and Communications*, 2013.