Battery-Aware Transformations in Mobile Applications

Jürgen Cito
University of Zurich
Zurich, Switzerland
cito@ifi.uzh.ch

Julia Rubin
MIT
Cambridge, MA, USA
mjulia@mit.edu

Phillip Stanley-Marbell
MIT
Cambridge, MA, USA
psm@mit.edu

Martin Rinard
MIT
Cambridge, MA, USA
rinard@mit.edu

ABSTRACT

We present an adaptive binary transformation system for reducing the energy impact of advertisements and analytics in mobile applications. Our approach accommodates both the needs of mobile app developers to obtain income from advertisements and the desire of mobile device users for longer battery life. Our technique automatically identifies recurrent advertisement and analytics requests and throttles these requests based on a mobile device's battery status. Of the Android applications we analyzed, 75% have at least one connection that exhibits such recurrent requests. Our automated detection scheme classifies these requests with 100% precision and 80.5% recall. Applying the proposed battery-aware transformations to a representative mobile application reduces the power consumption of the mobile device by 5.8%, without the negative effect of completely removing advertisements.

CCS Concepts

•**Software and its engineering** → **Software performance;** •**General and reference** → *Metrics;*

Keywords

Energy efficiency, battery lifetime, mobile advertisements, program analysis.

1. INTRODUCTION

Mobile devices have limited battery capacity, which restricts their utility for users. Prior research has identified mobile advertisement and analytic (A&A) services as a significant contributor to battery drain [11, 12]. Removing A&A services completely can reduce the energy usage of a mobile device by up to 16% [8].

However, A&A services are an important part of the app ecosystem: they are a popular way for mobile app developers to earn revenue (advertisements) and to gain insight into user behavior (analytics). Thus, existing approaches, which remove ads altogether, are insufficient. To balance the desire of developers to include numerous A&A requests in their apps with the desire of the users to maximize the battery lifetimes of their devices, we propose an approach that dynamically adapts the rate of A&A requests according to the state of a mobile device's battery.

Battery-Aware Transformations. We present an approach to balance the interests of mobile app developers with the interests of mobile device users. The main idea behind our approach is to automatically identify recurrent A&A requests in existing applications, detect their frequency and modify application binaries so that applications can adapt the A&A frequency to the current battery state.

Our initial results show that at least 75% of the A&A requests in Android applications are recurrent. These recurrent requests can be detected in an automated manner: the detection scheme proposed in this work classifies the recurrent requests with 100% precision and 80.5% recall. Our preliminary experiments applying the battery-aware transformation to a representative Android application from the Google Play Store reduce the average whole-system power dissipation on a Nexus 4 mobile phone by 5.8%, without the effect of removing ads completely, which is undesirable for developers.

Looking forward, there are several possible models of applying the proposed approach. One option is to decrease the rate of recurrent requests in a *linear* manner as the battery discharges. Another option is to activate the rate reduction when the phone enters a *low power mode*: in most modern mobile operating systems, such as Android and iOS, low-power modes are activated when the battery status reaches a prescribed level, say, 20%. These modes are designed to reduce energy consumption in exchange for reduced functionality [3, 13]. For example, on iOS 9, the low-power mode user interface states that when active, *"mail fetch, background app refresh, automatic downloads, and some visual effects are reduced or disabled."* [3]. Our approach is thus aligned with the vision and intentions of mobile operating system designers.

Contributions. This paper introduces a technique for improving energy consumption in apps that use A&A services. The proposed techniques balance the interests of both the mobile app developer (ads and analytics) and the mobile device user (longer battery life). Our initial implementation and results make four contributions to the state of the art:

1. **Empirical evidence for recurrent requests** in Android apps: our analysis shows that 75% of the apps we studied have *at least one* recurrent request.

ASE '16 September 3–7, 2016, Singapore

DOI: 10.475/123_4

2. **An algorithm for detecting recurrent requests** in Android apps through dynamic analysis: our algorithm determines the period of recurrent requests and achieves 100% precision and 80.5% recall.

3. **A battery-aware transformation** for dynamically transforming Android binaries to throttle recurrent requests based on the current battery status.

4. **Initial evaluation** for savings achieved by the battery-aware transformation: our experiments show that applying the proposed transformation on a single application can reduce the power consumption of a device by 5.8%.

These contributions demonstrate the feasibility and usefulness of this novel type of approach: unlike earlier work, it is designed to strike the balance between the needs of mobile application users and the financial incentives of mobile application developers.

Structure of the paper. The rest of the paper is organized as follows. Section 2 presents an empirical study to determine the prevalence of recurrent A&A requests in Android apps. Section 3 elaborates our proposed approach, divided into profiling, frequency analysis, and battery-aware transformation. In Section 4, we apply our approach to a use case application and report on the energy savings. Section 5 positions this paper within the related work. Section 6 describes the limitations of our work with an outlook to the future. We conclude with a summary in Section 7.

2. FREQUENCY OF REQUESTS

We first conduct an empirical study to investigate the frequency of recurrent A&A requests in Android applications.

2.1 Study Design

As the subjects of our study, we downloaded the 30 most-popular applications available on the Google Play store as of February 2016. We instrumented application binaries to log information about A&A connection statements that these applications make. We used the **dex2jar**[1] toolsuite to extract the **jar** file from the apps' **apk**s and Soot [9] (as implemented in FlowDroid [4]) to create instrumented versions of the original applications.

Our study considered as a connection any statement listed in Table 1.

Table 1: Connection statements.

Class/Interface	Method
java.net.URL	openConnection
java.net.URLConnection	connect
org.apache.http.client.HttpClient	execute
org.jsoup.helper.HttpConnection	connect
java.net.HttpURLConnection	getOutputStream
java.net.Socket	getOutputStream

We also included all sub-classes of those listed in the table. We further analyzed package names of each class that issues a connection statement to identify those that belong to the known A&A libraries.

For each of the identified A&A connection statements we inserted a corresponding **print** statement whose output is captured by the Android system logging facilities. We logged the outgoing request's type and time of occurrence.

We then transformed the **jar** file back into an **apk** using the **dex2jar** toolsuite and signed this **apk** using the standard **jarsigner** tool from the Java JDK.

We installed the instrumented applications on a Nexus 4 mobile device running Android 4.4.4, excluding applications where the Soot-based binary transformations failed. We believe that these failures occur because the apps in question might use language constructs that are not supported by the current version of Soot. We also excluded applications that required user-specific input such as login credentials and chat partners. The remaining eight applications are listed in the first column of Table 2.

We ran each application for 30 minutes while connected to WiFi with the device stationary. We did not interact with the applications. At the end of the run, we obtained a file which logs all connection issued by the application. We refer to that file as a *connection log*.

2.2 Study Results

We manually analyzed the produced connection logs for each application to classify requests as recurrent or non-recurrent. The results of this analysis are summarized in columns 3 and 4 of Table 2.

Our study shows that 75% of the analyzed applications have at least one recurrent A&A connection that is activated without the user interacting with the application. We also observed that one application has six such connections. On average, applications have 6.25 A&A connections, out of which 1.7 are recurrently activated without any user interaction.

Additionally, we manually extracted the intervals between recurrent requests by carefully inspecting the connection log for each request. On average, recurrent requests occur every 49.6 seconds (column 5 in Table 2).

Summary: Our empirical study shows that 75% of applications have at least one recurrent request that is activated without any user interaction with an application. On average, applications have 1.7 such requests, which occur every 49.6 seconds.

3. PROPOSED APPROACH

Inspired by the study described in Section 2, we propose an automated approach for modifying application binaries to adjust the frequency of recurrent A&A request according to the battery state. Our approach consists of three main steps shown in Figure 1: Profiling, frequency analysis, and battery-aware transformation. These steps are described in Sections 3.2, 3.3 and 3.4. Before that, we give a more formal definition of recurrent requests.

3.1 Recurrent Requests

We denote outgoing connection statements in an app's bytecode with $c_1, \ldots, c_n$. Table 1 lists the types of connection statement we consider. Whenever one of the connection statements cause a request at runtime, the identity of the connection statement along with a timestamp t is recorded. We assume that there is an observation at every possible timestamp $t \in \mathbb{N}$. As such, there is a binary sequence

[1]https://github.com/pxb1988/dex2jar

Table 2: Recurrent A&A requests in Android applications.

			Manual Analysis		Automated Analysis	
APK Name	Size	# AA	# Rec.	ρ	Precision	Recall
com.amazon.mShop.android.shopping	27M	12	1 ✓	43	100% (0/0)	0% (0/1)
com.vlcforandroid.vlcdirectprofree	1.8M	2	1 ✓	30	100% (1/1)	100% (1/1)
com.socialping.lifequotes	4.9M	1	1 ✓	60	100% (1/1)	100% (1/1)
com.surpax.ledflashlight.panel	5.8M	13	0	-	-	-
net.zedge.android	9.4M	11	6 ✓	45	100% (5/5)	83% (5/6)
com.lionmobi.powerclean	3.4M	6	0	-	-	-
com.cynomusic.mp3downloader	3.2M	3	1 ✓	60	100% (1/1)	100% (1/1)
com.vysionapps.faceswap	5.5M	2	1 ✓	60	100% (1/1)	100% (1/1)
Average	7.6M	6.25	1.7	49.6	100%	80.5%

$\mathcal{X}_{c_i} = \{(t, x(t)) : t \in \mathbb{N}\}$ for every connection statement c_i. For a given connection statement c_i, $x(t) = 1$ if the connection issued a request at some point in time t, and $x(t) = 0$ if the connection statement issued no request at time t.

Definition 1. *(Recurrent Request) A time series $\mathcal{X}_{c_i} = \{x(t)\}$ for a connection statement c_i is said to be recurrent if there exists some $\rho \in \mathbb{N}$ such that $x(t + \rho) = x(t)$ for all t of $\mathcal{X}_{c_i}$. ρ is called the period of $\mathcal{X}_{c_i}$.*

We define a subset $\mathcal{T}^+_{c_i} = \{t : x(t) = 1\}$ as the collection of timestamps of *positive events*, i.e., when connections have issued requests in $\mathcal{X}_{c_i}$. Given two consecutive timestamps t_i and t_{i+1} in the set $\mathcal{T}^+_{c_i}$, we consider a series to be recurrent with a period ρ with respect to a threshold $\tau \in [0..1]$, if

$$\frac{|(t_{i+1} - t_i) - \rho|}{\rho} < \tau.$$

3.2 Profiling

For the profiling step, we follow the procedure described in detail in Section 2.1. The input for this step is an `apk` file. We instrument the bytecode and annotate all connection statements listed in Table 1 (including its sub-classes). We exercise the application for 30 minutes without any interaction. The output is a connection log that specifies type and time-stamps of the A&A connection statements made by the application.

3.3 Frequency Analysis

For frequency analysis, we develop an automated algorithm that identifies recurrent requests by analyzing connection logs. We detect recurrent requests using the procedure outlined in Algorithm 1. The algorithm takes as input the series of timestamps when a request has been issued for a connection $t_i, \ldots, t_n$ in $\mathcal{T}^+_{c_i}$. From this series, the algorithm computes a new series of deltas between each timestamp, $\delta^+_{c_i} = \{t_{i+1} - t_i \mid 0 > i > n\}$.

The algorithm first generates the request series by computing a moving average of request timestamps using a window size w, to group subsequent requests together. Next, the algorithm calculates the deltas, $\delta^+_{c_i}$, between subsequent requests. From this distribution of $\delta^+_{c_i}$ values, the algorithm uses a maximum-likelihood estimator (MLE) to estimate the distribution's parameters. The algorithm then computes the relative standard deviation (rsd_{deltas}) and compares it against a threshold τ. If the rsd_{deltas} is below the threshold, the algorithm considers the series as containing recurrent requests and return the estimated mean μ_{deltas}.

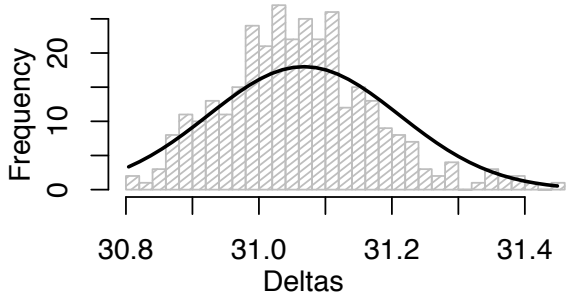

Figure 2: Distribution of deltas (differences between two consecutive timestamps) between requests. The line is a fitted normal distribution parameterized by values retrieved through maximum-likelihood-estimation (MLE) in R package `fitdist`.

Figure 2 shows a real-life example of such a series of deltas between requests of the connection statements, residing in the package `com.google.android.gms.b.os`, in the *VLC Direct* Android app. We observed that the distribution of deltas associated with connections having recurrent requests in many of our subject applications was similar to the distribution observed in Figure 2.

Algorithm 1 Detecting Recurrent Requests.

```
1: procedure DetectRecurrentRequest(series, w, τ)
2:     series ← movingAverage(series, w)
3:     deltas ← calculateDeltas(series)
4:     (μ_deltas, σ_deltas) ← MLE(deltas)
5:     rsd_deltas ← σ_deltas / μ_deltas
6:     if rsd_deltas < τ then
7:         return μ_deltas
8:     else
9:         return -1
```

3.3.1 Evaluation

We assess the quality of our frequency analysis by applying the procedure of Algorithm 1 to all connection logs from our subject applications, using a moving average window of $w = 5$ seconds and a threshold $\tau = 0.25$. We compare the results against the ground truth established in the manual analysis described in Section 2.

Columns 7 and 8 in Table 2 present the resulting precision and recall for the comparison. *Precision* refers to the fraction of connections correctly identified as recurrent among those

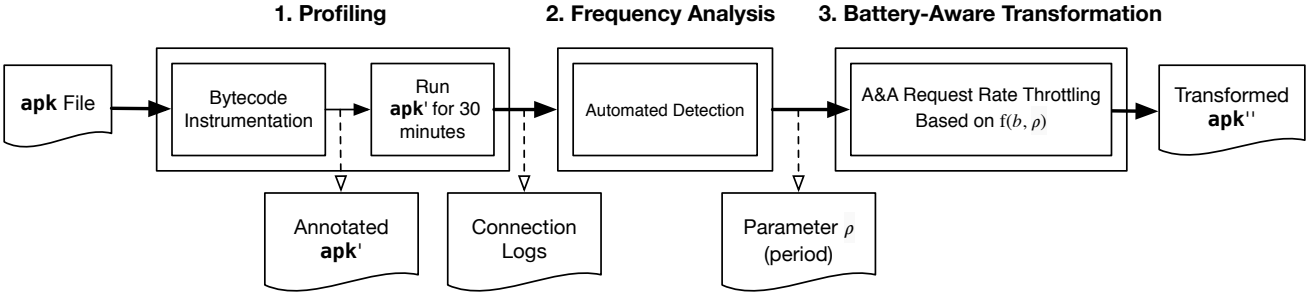

Figure 1: Approach overview illustrating all stages from original apk to Battery-Aware Transformed version of the apk

reported by the automated analysis. *Recall* refers to the fraction of connections correctly identified as recurrent from the expected ground truth set. On average, the algorithm achieves 100% precision and 80.5% recall.

There were only a small number of cases where the algorithm could not correctly classify connections. Manual inspection of request traces led us to believe that these cases were the result of more than one process in an app executing the same statement (e.g., a statement in a library), and as a result generating requests. The resulting time series from such a process does not correspond to our current notion of recurrent requests. More in-depth analysis is required to either adjust the definition or detection mechanism for recurrent requests.

Summary: Our automated mechanism for detecting recurrent A&A requests was able to achieve 100% precision and 80.5% recall with respect to the ground truth established in manual analysis.

3.4 Battery-Aware Transformation

The idea behind our battery-aware transformation is to transparently modify apps to limit the rate at which they perform recurrent A&A requests. Our goal in performing these modifications is to reduce the number of times a mobile device's network interface is activated. Because the network interface is an important contributor to power dissipation in mobile devices [15], reducing communication should improve whole-system energy-efficiency.

As we want to balance the interests of app developers against those of mobile device users, we modulate the fraction of recurrent requests that we remove, based on the device's battery level. We modify apps so that, at full battery charge, the code we insert into app bytecode permits all recurrent requests. Near complete battery depletion, the inserted code permits no recurrent requests.

3.4.1 Transformation Models

We consider a series of recurrent requests $\mathcal{R}_{c_i,\Delta t} = \{r_i, \ldots, r_n\}$ in connection c_i within a time span Δt. The series $\mathcal{R}_{c'_i,\Delta t}$ of a transformed connection c'_i is a subset $\mathcal{R}_{c'_i,\Delta t} \subseteq \mathcal{R}_{c_i,\Delta t}$ of the recurrent requests in the original connection over the same time span Δt. There are two ways one could achieve this request rate limiting. The first option would be to skip (every N) requests to achieve a lower number of requests over the time span Δt. The other option is to insert a delay before or after every connection. We investigate the delay option, as it allows for more fine-grained control over controlling the rate of requests.

We introduce a delay that is a function $f(b, \rho)$ of battery status b and the period of the recurrent request ρ. We use a transformation model that linearly increases the delay as battery status decreases (Equation 1). We include the constant factor c to allow for adjustment in linear increase or decrease of the delay:

$$f_{linear}(b, \rho) = \frac{1}{c * b} * \rho \tag{1}$$

Following the notion of a *low power mode* we discussed in the introduction, a possible transformation model would be to throttle resource consumption when hitting a specific lower battery status, e.g., 20% (see Equation 2, c again, is a constant factor applied to the period ρ).

$$f_{LowPowerMode}(b, \rho) = \begin{cases} \rho * c, & \text{if } b \leq 0.2 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

3.4.2 Application Transformation

We again use Soot to instrument the application binaries and insert a delay following a transformation model. In the following, we outline the method `BatteryAwareTransformer`:

- The transformation is implemented through a `BodyTransformer` in Soot, in which all methods in a binary are iterated.
- The method provides a collection of *units*, which roughly correspond to a line of bytecode (i.e., invocations, assignments, etc.).
- Given a map of detected connections paired with their periods ρ as parameters, the transformer checks whether (a) the unit contains an invocation that is a connection, and (b) it is contained in the map of detected connections.
- If a given unit is identified as one of the detected connections, we inject a `Thread.sleep()` invocation[2] with a chosen transformation model $f(b, \rho)$ as its parameter.
- Since `Thread.sleep()` may cause a `InterruptedException`, we also inject a trap with this exception that does nothing.

[2]More specifically: <java.lang.Thread: void sleep(long)>

4. CASE STUDY: ENERGY SAVINGS

We apply the battery-aware transformation on the use case application *VLC Direct* to demonstrate potential energy savings. Since energy models on mobile devices suffer from limited accuracy [5], we assume a constant delay at a particular lower battery level. We thus apply Equation 2 as our battery saving transformation model.

To determine the difference in consumed energy, we measured the energy consumption of the application before and after battery-aware transformation. We ran each experiment for 30 minutes. We repeated the measurement five times for each version to minimize the possible impact of background noise. For this initial evaluation, we sampled energy consumption information from the device's fuel gauge [1, 2] at 1Hz. That is, we polled the **`/sys/class/power_supply/battery/uevent`** interface (fuel gauge) for the voltage and current, which was sufficient for an initial investigation of the energy consumption. We intent to look into more sophisticated measures of energy as part of future work.

Table 3: Energy Consumption (average power dissipation) of VLC Direct before and after transformation

(mW = Milli-Watts)	Before Approach ($\rho = 30$)	After Approach ($\rho = 60$)
Run 1	1138 mW	1056.87 mW
Run 2	1147.87 mW	1098.4 mW
Run 3	1085.26 mW	1067.57 mW
Run 4	1198 mW	1088.79 mW
Run 5	1200.99 mW	1120.99 mW
Average	1154 mW	1086.52 mW
StdDev	47.88	25.36
RSD	4.14%	2.33%
Energy Saving	**Absolute Savings 67.48 mW**	**Savings in % 5.86%**

The results of the energy evaluation case study are presented in Table 3. The values shown for each run represent the average power dissipation (total power consumption divided by time of measurement). We can see that the application after the battery-aware transformation was able to decrease energy consumption by 5.86%. This result has to be seen relative to the potential power savings: removing ads completely yields 16% of average power savings [8], while our approach strikes the balance between the needs of mobile application developers and users. Furthermore, in lower power mode (i.e., low battery status, around 20%) even smaller savings in energy consumption can make a difference. We thus view this initial result as promising and expect even larger saving when the approach is applied to *all* applications running on the same device.

Summary: Applying the battery-aware transformation to a use case application yielded 5.86% of savings in energy consumption.

5. RELATED WORK

There has been a multitude of work researching energy consumption in mobile applications. In the following, we focus on work that has investigated the impact of advertisement on mobile applications and approaches of program analysis and transformation to reduce energy consumption.

Impact of Advertisment on Mobile Applications: Work by Gui et al. investigated the impact of ads on different resources, including energy consumption [8]. In a study considering similar aspects, Pathak et al. examined where the energy is consumed in different apps, and found that third-party advertisement APIs is a large factor [11]. Work by Prochkova et al. specifically explores the impact of ads in mobile games, and found that applications that request information on ad servers more frequently consume more energy [12]. Work by Chen et al. investigates the upper bound of energy savings by prefetching ads [6]. Previous work by Rubin et al. studied the occurrence of requests that have no effect on user-observable functionality, and found that ads are a substantial part of these "covert communications" [14].

Program Analysis and Transformation to Reduce Energy Consumption: Li et al. use static analysis to reduce the energy consumption of apps by automatically bundling subsequent HTTP requests [10]. Recent work by Gui et al. explores methods for measuring and predicting ad related energy consumption based on static analysis [7]. While these approaches remove ads completely to analyze the impact of ads on mobile resources, in this paper, we propose to throttle the rate at which the ads are retrieved based on the current battery status.

Other works in the area of program analysis and transformation have looked at improving display energy consumed by apps [17, 16]. In contrast, the focus of our work is on recurrent requests, regardless of whether they are manifested in display updates or not.

6. LIMITATIONS AND FUTURE WORK

*E*mpirical Study: Using only eight subject applications might limit the generalizability of our study if the selected applications are not representative. We try to mitigate this problem by choosing an unbiased sample. We selected our subject applications from the most popular applications from the app store.

We did not interact with applications when investigating connection patterns during the dynamic analysis. This limits the potential findings of further recurrent request patterns. Furthermore, we did not include apps that require any kind of user interaction (especially logins) to become fully usable. Thus, if anything, we are likely underreporting the prevalence of recurrent requests. As a follow up to these explorations, we plan to investigate how the phenomenon of recurrent requests manifests in more diverse types of applications and possible sequence of user interactions.

*E*nergy Analysis: We chose one use case application to demonstrate the potential energy savings of the novel approach. The result of this early investigation is the basis for further exploration in this area. In a future study we plan to evaluate the approach with more applications to see how much energy can possibly be saved for applications from different categories (e.g., games and utility apps), different ad-frameworks (e.g., Google AdMob[3] or Unity Ads[4]), different ad-types (just text, photos or animations) and connection types (phone connected via 3G, 4G, or Wi-Fi).

[3]https://www.google.com/admob/
[4]https://unity3d.com/services/ads

Modifying the binaries to introduce **print** statements might affect the energy footprint of the studied application. However, these modified statements were present in both versions (before and after transformation) of the application. If anything, we are underestimating the potential energy savings.

To avoid unexpected results, we performed all experiments of dynamic nature on the same device (Nexus 4 with Android 4.4.4) and at the same location. To demonstrate energy savings, we also ran the experiment five times to reduce possible background noise. We use the fuel gauge of the phone to retrieve the necessary energy information at 1Hz. Since our energy analysis is comparative, we believe the accuracy provided by this method suffices. However, for future explorations of this space, we intend to use a more fine-grained and accurate approach such as the Monsoon Power Meter[5].

*F*uture Explorations: In general, we want to continue studying the compromise between the usage of advertisement and analytics and energy consumption more in-depth. One possible route of investigation is Gamification (for instance assigning "Energy Badges") to motivate developers to enable our approach. Another possible route of study would be to investigate game-theoretical considerations with mobile developers and users as actors within the mobile market space. Furthermore, the study of this compromise in itself could inform the design of future battery-aware APIs.

7. CONCLUSION

We presented early work on an automated approach that addresses the trade-off of using advertisement and analytics services in mobile applications (to increase the revenue of developers and provide them with insights into user behavior) and reducing energy consumption (to increase the usability of the device). Our approach identifies advertisement and analytics connections that contain recurrent requests to outgoing servers and throttles the number of requests over time, based on the current battery state. The results presented in this paper show potential for further research. Our proposed approach is intended to start a conversation about the conflict between the objectives of mobile developers and mobile users. The next steps in this exploration will include a more fine-grained view on the energy saving potential, as well as game-theoretical considerations to find a Pareto-optimal solution of our aforementioned multi-objective optimization problem. We could imagine the end result of our investigation to be implemented by controlling authorities (e.g., application stores) in the form of gamification ("Energy Badge") or to inform ecosystem developers to build battery-aware APIs.

8. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave) and the CHOOSE Forum[6].

9. REFERENCES

[1] *Measuring Device Power*, 2016 (accessed April 29, 2016). https://source.android.com/devices/tech/power/device.html.

[2] *Android Fuel Gauge Analysis Tool*, 2016 (accessed July 15, 2016). https://github.com/phillipstanleymarbell/android-fuel-gauge-uevent-analysis.

[3] Apple, Inc. iOS 9 low power mode. (accessed July 15, 2016). https://support.apple.com/en-us/HT205234.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA, 2014. ACM.

[5] J. Bornholt, T. Mytkowicz, and K. S. McKinley. The model is not enough: Understanding energy consumption in mobile devices. In *Hot Chips 24*, 2012.

[6] X. Chen, A. Jindal, and Y. C. Hu. How much energy can we save from prefetching ads?: energy drain analysis of top 100 apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems*. ACM, 2013.

[7] J. Gui, D. Li, M. Wan, and W. G. Halfond. Lightweight measurement and estimation of mobile ad energy consumption.

[8] J. Gui, S. Mcilroy, M. Nagappan, and W. G. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 2015.

[9] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. 2011.

[10] D. Li, Y. Lyu, J. Gui, and W. Halfond. Automated energy optimization of http requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[11] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 2012.

[12] I. Prochkova, V. Singh, and J. K. Nurminen. Energy cost of advertisements in mobile games on the android platform. In *6th International Conference on Next Generation Mobile Applications, Services and Technologies (NGMAST)*, 2012.

[13] Qualcomm. Snapdragon BatteryGuru. (accessed July 15, 2016). https://qualcomm.com/products/snapdragon/power-efficiency.

[14] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard. Covert communication in mobile applications (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015.

[15] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[16] P. Stanley-Marbell, V. Estellers, and M. C. Rinard. Crayon: saving power through shape and color approximation on next-generation displays. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, page 11, 2016.

[17] M. L. Vásquez, G. Bavota, C. E. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk. Optimizing energy consumption of GUIs in Android apps: a multi-objective approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, 2015.

[5] http://www.msoon.com/LabEquipment/PowerMonitor

[6] http://www.choose.s-i.ch/