How Does Code Obfuscation Impact Energy Usage?

Cagri Sahin, Philip Torquist, Ryan McKenna, Zachary Pearson, and James Clause

Computer and Information Sciences Department
University of Delaware
Newark, DE 19716

Email: {cagri, philip, mckennar, zpearson, clause}@udel.edu

Abstract—Software piracy is an important concern for application developers. Such concerns are especially relevant in mobile application development, where piracy rates can approach 90%. The most commonly used approach by mobile developers for preventing piracy is code obfuscation. However, the decision to apply such transformations is currently made without regard to the impacts of obfuscations on another area of increasing concern for mobile application developers: energy consumption. Because both software piracy and battery life are important concerns, mobile application developers must strike a balance between protecting their applications and preserving the battery lives of their users’ devices. To help them make such choices, we conducted an empirical study of the effects of 18 code obfuscations on the amount of energy consumed by executing a total of 15 usage scenarios spread across 11 Android applications. The results of the study indicate that, while obfuscations can apply such transformations is currently made without regard to their impacts on another area of increasing concern for mobile application developers: energy consumption. As a result, an obfuscated application may consume an excessive amount of power, draining the battery and causing users to leave poor reviews or request refunds. Because both software piracy and battery life are important concerns, mobile application developers must strike a balance between protecting their applications and intellectual property and preserving the limited battery power of the devices where their applications will execute. A major obstacle to striking an appropriate balance between these concerns is a lack of information about how high level changes to an application impact its energy consumption. As a result developers must either make a poorly informed choice, or more commonly, choose not to choose by using an obfuscation tool’s default configuration. Unfortunately, these approaches often result in applications that either consume more energy than necessary or are not protected as well as they could be.

1. INTRODUCTION

Software piracy is an important concern for application developers. A 2011 study conducted by the Business Software Alliance found that over 40% of software is pirated, resulting in a potential loss of over $63 billion. Such concerns are especially relevant in mobile application development, where piracy rates can approach 90%. The most commonly used approach by mobile developers for preventing piracy is code obfuscation—making the code of their applications more difficult for a human to understand. Both Microsoft and Google strongly recommend that developers obfuscate their applications. Google has even gone so far as to integrate obfuscation into the standard Android build system. In practice, there are many different types of transformations (e.g., renaming variables and methods; merging, splitting, and reordering code; complicating control and data flow, etc.) that are used to successfully obfuscate code. However, the decision to apply such transformations is currently performed without regard to their impacts on another area of increasing concern for mobile application developers: energy consumption. As a result, an obfuscated application may consume an excessive amount of power, draining the battery and causing users to leave poor reviews or request refunds.

1. The impacts of obfuscation on battery life are unlikely to be meaningful to mobile application users.
2. Because both software piracy and battery life are important concerns, mobile application developers must strike a balance between protecting their applications and preserving the limited battery power of the devices where their applications will execute. A major obstacle to striking an appropriate balance between these concerns is a lack of information about how high level changes to an application impact its energy consumption. As a result developers must either make a poorly informed choice, or more commonly, choose not to choose by using an obfuscation tool’s default configuration. Unfortunately, these approaches often result in applications that either consume more energy than necessary or are not protected as well as they could be.

To address the lack of information available to developers, we have conducted an empirical study investigating the energy impacts of applying different code obfuscations. At a high-level, this study is focused on addressing two major questions: (1) How do different obfuscations alter the overall energy consumption of an application?, and (2) Are the impacts of obfuscations likely to be meaningful for mobile application users? To answer these questions, we used 4 commonly used obfuscation tools to create a total of 198 obfuscated versions of 11 Android applications. We executed each obfuscated version multiple times with one or more different user scenarios and recorded the amount of energy that was consumed. In total, we ran 8,550 executions and collected over 3.2 GB of experimental data. We then performed a statistical analysis of the collected data to investigate the impacts of the obfuscations on energy usage and answer our research questions.

The results of our study show that:

1. Obfuscations can, and often do, impact the energy usage of applications with statistical significance.
2. Obfuscations can both increase and decrease energy usage, but they are much more likely to increase energy usage.
3. The magnitude of the impacts of obfuscations is comparable to the magnitude of the impacts of other code level changes such as applying refactorings.
4. The differences between the impacts of the considered obfuscations on energy usage are not statistically significant.
5. The impacts of obfuscation on battery life are unlikely to be meaningful to mobile application users.

As such, we believe that these results are positive news for both mobile application users and mobile application developers. Developers can protect their applications without impacting the battery life of the devices where their applications execute.
The remainder of this paper is organized as follows: Section II describes the methodology of our study including our subjects and experimental procedure. Section III presents and discusses the results of the study including potential threats to its validity. Finally, Sections IV and V discuss related work and present our conclusions and future work.

II. EMPIRICAL STUDY

This section describes the details of our study design, including our considered variables, considered obfuscations, and data collection protocol. In planning this work, we followed the recommended guidelines for empirical study design [1]. All of our experimental applications, artifacts, and summary data are publicly available. Our raw experimental data is too large to host publicly, but it is available upon request.

A. Experimental Variables

In this study, we considered one dependent variable, the amount of energy consumed by the execution of an application, and one independent variable: the obfuscation applied to the application.

To isolate the impacts of changing our independent variable on our dependent variable, it is necessary to control for the effects of several extraneous variables (e.g., unnecessary changes in the considered application’s code and the inputs that are used to drive the application). The remainder of this section describes how we controlled for such extraneous variables.

1) Controlling for extraneous changes in an application’s code: In many cases, obfuscations are not formally specified. Because of this, different tools may use the same name to refer to different sequences of code changes. For example, many obfuscation tools provide a transformation called “string encryption.” At a high level, all of these transformations perform the same operation: encrypting the constant strings in an application so that they cannot be easily understood. However, the specific encryption algorithm used can vary greatly. This flexibility in nomenclature can be a potential source of bias and a potential source of confusion in interpreting the results of the study. If we compared the impacts of obfuscations that were inconsistently applied, we would essentially be comparing different transformations. Similarly, if a developer would apply a substantially different set of code edits that happen to share the same name as one of the obfuscations that we considered, the results that they observe could be drastically different than what we observed.

In order to avoid these potential problems, we ensured that all obfuscations were applied in a consistent, repeatable, and well documented manner. To accomplish this, we relied on several commonly used obfuscation tools (see Section II-D). By using preexisting, automated tools, we are ensuring that the changes we are making to our considered applications are the same changes that a developer would apply if they applied the same obfuscations using the same tool.

2) Controlling for inconsistencies in driving an application: In general, mobile applications are interactive and event-driven. They accept input, either from a user or from a sensor, perform some computation, and generate a response. In our experiments, this interactive nature can introduce a potential source of bias as it is difficult to manually reproduce a given execution exactly. For example, a user can often repeatedly 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. Although such differences may seem inconsequential, they may lead to observed differences in energy consumption that are not due to changing our independent variable, but rather to differences in how the application is driven. In order to prevent such bias, it is necessary to be able to reproduce deterministically a given sequence of actions with great fidelity; capture/replay tools provide this functionality.

Capture/replay tools are designed to allow for the deterministic replay of a sequence of recorded events. Conceptually this is accomplished by wrapping an application to insulate it from its environment. When capturing, the wrapper records all of the events that are passed to the application from the environment. When replaying, the wrapper replaces the environment and passes the recorded events to the application. Because precise timing information is recorded during the capture process, there is very little variability in when events are passed to the application during replay. Hence, when using a capture replay tool, any observed variations in energy usage are more likely to be the result of the obfuscations used rather than inconsistencies in driving the application. We choose to use RERAN [2] as our capture/replay tool, because it is designed to record and replay Android applications.

B. Considered Applications

As the applications for our study, we used popular, easily accessible Android applications. We selected Android applications for several reasons. First, Android application developers typically care about both the security of their intellectual property and the energy efficiency of their applications. Second, there are many existing obfuscation tools that specifically target Android applications, or, more generally, operate on Java code, that we can use. Third, the source code of many Android applications is freely available, allowing us to easily create many different obfuscated versions. Finally, we have extensive infrastructure to run Android applications and measure their energy usage.

Table I lists the specific applications that we selected. The first two columns, Application and Description, list the name of each application and a brief description of its functionality.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>LoC</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnkiDroid</td>
<td>Flashcard application</td>
<td>44,913</td>
<td>2.4</td>
</tr>
<tr>
<td>Calculator</td>
<td>Default Android calculator</td>
<td>1,427</td>
<td>2.6</td>
</tr>
<tr>
<td>Calendar</td>
<td>Default Android calendar</td>
<td>43,715</td>
<td>1.4</td>
</tr>
<tr>
<td>Clock</td>
<td>Default Android clock</td>
<td>13,477</td>
<td>1.0</td>
</tr>
<tr>
<td>DailyMoney</td>
<td>Daily financial tracker</td>
<td>8,723</td>
<td>0.4</td>
</tr>
<tr>
<td>FrozenBubblePlus</td>
<td>Bubble popping puzzle game</td>
<td>7,517</td>
<td>0.2</td>
</tr>
<tr>
<td>Nim</td>
<td>Mathematical strategy game</td>
<td>1,475</td>
<td>0.8</td>
</tr>
<tr>
<td>OIFileManager</td>
<td>File manager</td>
<td>7,200</td>
<td>0.7</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td>Sudoku game</td>
<td>6,079</td>
<td>0.2</td>
</tr>
<tr>
<td>SkyMap</td>
<td>Astronomy application</td>
<td>10,921</td>
<td>0.7</td>
</tr>
<tr>
<td>Tomdroid</td>
<td>Note taking application</td>
<td>7,955</td>
<td>0.6</td>
</tr>
</tbody>
</table>

https://bitbucket.org/udse/obfuscation-study
The third column, LoC, shows the application’s number of lines of code and the final column, size, shows the size of the application’s compiled application package file (APK). The LoC measurement includes only the application itself, while the size measurement includes both the application and its necessary libraries. Because our considered obfuscation tools obfuscate both the application and its libraries, even when the source of such libraries is unavailable, we chose to report both measures to give a better understanding of the amount of code that is being obfuscated.

We choose these specific applications for several reasons. First, they are representative of a wide variety of common application types (e.g., games, study aids, productivity tools, etc.). Second, they are popular and widely used. For example, Calculator, Calendar, and Clock are part of the default Android installation. Finally, they are supported by RERAN. Although RERAN is generally effective at replaying user inputs such as touch events, it does not support replaying network connections or other sensor readings (e.g., GPS). As such we were unable to include applications that depend on these types of inputs.

Note that in order to experiment with these applications successfully, we needed to modify them slightly. Primarily, the modifications were made to their build systems so that we could automate the obfuscation processes, but in some cases we also needed to modify the application’s source code to remove sources of randomness that are not handled by RERAN (e.g., we modified the random number generator to use a fixed seed).

C. Considered Usage Scenarios

As we mentioned previously, our considered applications are driven primarily by user input. To create the inputs necessary for driving the applications, we examined each application and created one or more usage scenarios. Our goal in creating these scenarios was to capture what we believe to be typical usage patterns for the application (i.e., actions that users are likely to perform). By focusing on typical scenarios rather than scenarios designed to maximize other metrics such as coverage, we are able to gain a better understanding of the impacts of obfuscations on a user’s daily interactions with their mobile device.

To convert the usage scenarios into replay-able executions, we manually performed the actions contained in each scenario while using RERAN’s recording tool. In total, we created 15 replay-able usage scenarios for our applications: two for AnkiDroid, Clock, DailyMoney, and SkyMap; and one for Calculator, Calendar, FrozenBubblePlus, Nim, OIFileManager, OpenSudoku, and Tomdroid. As a sanity check, we then verified that RERAN could accurately replay the recorded scenarios by observing the replayed executions.

Table II shows the specific usage scenarios that we created. The first two columns, Application and Name, show the application that is used in the scenario and, if there is more than one scenario for an application, a distinguishing name. For example, there are two scenarios for AnkiDroid, AnkiDroid: New Deck and AnkiDroid: Tutorial Deck. The third column, Description, provides a brief description of what user actions are performed during the scenario. For example, during the

<table>
<thead>
<tr>
<th>Application</th>
<th>Name</th>
<th>Description</th>
<th>Dur. (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnkiDroid</td>
<td>New Deck</td>
<td>Creates a new slide deck containing 5 cards</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>Tutorial Deck</td>
<td>Reviews the 20 cards in the tutorial deck</td>
<td>60</td>
</tr>
<tr>
<td>Calculator</td>
<td></td>
<td>Does several basic arithmetic calculations</td>
<td>57</td>
</tr>
<tr>
<td>Calendar</td>
<td></td>
<td>Adds a new event, searches for it, deletes it</td>
<td>108</td>
</tr>
<tr>
<td>Clock</td>
<td>Stopwatch</td>
<td>Runs the stopwatch for 10 seconds</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Timer</td>
<td>Runs a 10 second countdown timer.</td>
<td>18</td>
</tr>
<tr>
<td>DailyMoney</td>
<td>Add Detail</td>
<td>Enters two transactions.</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>View Lists</td>
<td>Views details for the past week, views details for the past year</td>
<td>16</td>
</tr>
<tr>
<td>FrozenBubblePlus</td>
<td></td>
<td>Plays the first level.</td>
<td>35</td>
</tr>
<tr>
<td>Nim</td>
<td></td>
<td>Plays three rounds with increasing difficulty levels</td>
<td>74</td>
</tr>
<tr>
<td>OIFileManager</td>
<td></td>
<td>Opens a file. Navigates directories.</td>
<td>17</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td></td>
<td>Completes a single “easy” Sudoku grid.</td>
<td>272</td>
</tr>
<tr>
<td>SkyMap</td>
<td>Find Mars</td>
<td>Sets time to a fixed past date, searches for Mars</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>Move Zoom</td>
<td>Arbitrarily zooms in and out, and moves along the map.</td>
<td>64</td>
</tr>
<tr>
<td>Tomdroid</td>
<td></td>
<td>Creates a note, searches for text, opens note, deletes note</td>
<td>173</td>
</tr>
</tbody>
</table>

AnkiDroid: New Deck scenario, a new flash card deck is created and five flash cards are added to the newly created deck. Finally, the fourth column, Dur., shows the duration of the scenario (i.e., how long it took to perform all of the actions) in seconds (s).

D. Considered Obfuscations

1) Obfuscation Tools: Our choice of obfuscation tools was essentially arbitrary. Our only hard requirements were that an obfuscation tool could (1) obfuscate Android applications, and (2) be easily integrated into the standard Android build system. Because we are repeatedly obfuscating multiple applications, manually applying obfuscations is infeasible. Unfortunately, these requirements eliminated the majority of the free or open source Java obfuscation tools. While such tools can work well for standard Java software, they either introduce changes that result in invalid Android applications when the obfuscated class files are converted to the dex format or they cannot be integrated into the Android build system. The only free obfuscation tool that we found that met our requirements is Proguard 4.106 which is the obfuscation tool that is bundled with the Android Software Development Kit.

Because of the limited number of free tools that met our requirements, we also considered commercial obfuscation tools. Here we found tools that were more likely to fulfill our requirements. However, their trial or evaluation versions are often limited in functionality (e.g., they only obfuscate parts of an application, or do not support the full suite of configuration options). As such they are not suitable for our study. To obtain full-featured versions, we emailed the tool developers and asked if they would be willing to donate a copy of their obfuscation tool. As the result of this process, we obtained copies of three commercial obfuscation tools: Allatori 4.7,7 DashO 7.2,8 and Zelix KlassMaster 6.1.3 (ZKM).9

6http://proguard.sourceforge.net
7http://www.allatori.com
8http://www.preemptive.com/products/dasho
9http://www.zelix.com/klassmaster/
2) Obfuscation Configurations: After reading the manuals of Allatori, DashO, Proguard, and ZKM, we identified several, common high-level configurations or obfuscation types:

Control-flow (cf): produces “spaghetti logic” that is difficult or impossible to decompile by inserting branching and conditional instructions into the body of a method.

Rename (rename): renames packages, classes, methods, and fields to short meaningless names (e.g., “a”, “b”, etc.) and, if possible, moves classes into a single package.

Optimize (opt): removes unused classes, fields, methods, and attributes; performs simple bytecode optimizations (e.g., peephole optimizations); removes dead code.

String encryption (se): all constant strings in the application are replaced with an encrypted version; decryption methods are added so that string can be decrypted at runtime.

All (all): combines all other configurations supported by an obfuscation tool.

Note that, while the specific changes made by each tool for each configuration may vary (e.g., different string encryption algorithms may be used or branches may be inserted in different locations), from the point of view of an application developer, the results are essentially identical. In addition, not every configuration is supported by every tool. Table III shows which configurations are supported by which tools.

In Table III, the first column, Obfuscation tool shows our considered obfuscation tools and the remaining five columns, all, opt, rename, cf, and se, show our considered configurations. A checkmark (✓) indicates that a configuration is supported by a tool and a blank space indicates that a configuration is not supported. As the table shows, there are 18 supported combinations. In the remainder of the paper, we will refer to a combination of an obfuscation tool and an obfuscation configuration as an obfuscation.

E. Measurement

To measure the amount of energy consumed by executing an Android application, we used a custom-built energy measurement platform (EMP). Our EMP is based on a Nexus 4 with 8 GB of storage running Android version 4.3 (Jelly Bean). Instead of using the phone’s battery, we are powering the Nexus 4 directly from a 30 V, 5 A DC power supply (KORAD KA3005D). This ensure that the phone’s battery monitor senses a constant charge level and allows us to compare results across executions without having to worry about variations in the physical battery’s performance or the phone’s power-saving infrastructure.

To sample the voltage and current draw of the phone, the EMP uses two Arduino Unos, each equipped with an Adafruit INA219 High Side DC Current Sensor board. One Arduino is used to sense the voltage and current drawn from the DC power supply and the other is used to sense the voltage and current draw over the phone’s USB port. The INA219 reports voltage measurements in volts (V) and current measurements in milliamps (mA).

Monitoring the power drawn from the USB port as well as the power supply allows us to keep the phone connected to a desktop computer and to control it over the Android Debug Bridge (ADB) rather the disconnecting and reconnecting the USB before and after each execution, which is useful when running repeated trials.

Because our EMP measures power consumption via hardware that is external to the Nexus 4, it does not introduce any measurement overhead to the application. This is ideal, since it means that we do not have to factor out the amount of energy consumed by the monitoring infrastructure itself. However, it also means that the Nexus 4 and the Arduinos do not share a single clock that can be used to identify which samples occurred during an execution of interest. The desktop computer can solve this problem by providing the global clock necessary for performing synchronization. By having the desktop computer start the execution of interest over the USB connection using ADB, it is possible to discard power samples recorded before the start of the execution. Similarly, because the duration of the recorded scenarios are known, it is possible to identify samples that occur after the end of the execution.

While the EMP itself does not introduce measurement overhead, the replay infrastructure does. In order to replay a recorded execution, RERAN installs an application on the phone that injects events into the Android kernel’s device drivers. It is possible to account for energy consumed by this application by profiling its costs and subtracting them from the reported energy numbers. However, because we are concerned with energy consumption relative to a base line (i.e., before and after applying an obfuscation) rather than absolute numbers and because the energy costs are consistent across executions, we have not taken this step.

F. Procedure

Figure 1 shows, at a high-level, the procedure we followed in our study, divided into three main steps: Obfuscated Application Creation, Data Collection, and Post Processing. The remainder of this section describes these steps in detail.

1) Obfuscated Application Creation: The first step in our procedure was to create our set of obfuscated applications. To create the necessary obfuscated versions, we simply obfuscated each application (see Table I) using each obfuscation (see Table III). In total, we created 198 obfuscated applications: 11 applications with 18 obfuscations each.

2) Data Collection: To collect power usage data, we used RERAN to replay each usage scenario (see Table II) on the EMP, using both the unobfuscated and obfuscated versions of the scenario’s application. Each scenario was executed on each version of the application (unobfuscated and obfuscated) 30 times. While each scenario was executing, we recorded the current and voltage measurements sampled by the Arduinos.

To reduce the possibility of noise in the measurements, we turned on “airplane mode” and terminated all unnecessary

---

TABLE III. CONSIDERED OBFUSCATIONS.

<table>
<thead>
<tr>
<th>Obfuscation tool</th>
<th>all</th>
<th>opt</th>
<th>rename</th>
<th>cf</th>
<th>se</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allatori</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DashO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Proguard</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ZKM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

---

[10] Information about the effectiveness of these types of obfuscations can be found in related work (e.g., [3, 4]).

---

134
Fig. 1. Obfuscated application creation and data collection procedure.

applications and processes. Although we eliminated many possible sources of noise by carefully configuring the EMP, small fluctuations in energy consumption from execution to execution are still possible. Multiple runs (i.e., 30) allows for performing a statistical analysis on the impact of obfuscations that takes into account the possibility of such fluctuations.

In total, we ran 8,550 executions—15 scenarios \(\times\) (18 obfuscated versions + 1 unobfuscated version) \(\times\) 30 repetitions—which took \(\approx\) 177 hours (over one week) of continuous execution time and resulted in over 3.2 GB of raw power consumption data.

3) Post Processing: The final step in our procedure is to post-process the collected data by filtering it and converting it to a useable form. We first filtered the data to remove samples that occur either before or after the execution. We then converted the current and voltage samples to power measurements in watts by multiplying them together and then dividing by 1,000: watts (W) = volts (V) \(\times\) milliamperes (mA) \(\div\) 1,000. Finally, we converted the resulting power measurements to total energy consumption in joules by summing the results of multiplying each power measurement by the length of time between itself and the following sample: joules (J) = watts (W) \(\times\) seconds (s).

III. DATA ANALYSIS AND DISCUSSION

We refined our overall question of whether or not applying obfuscations can impact the energy usage of an application into the following specific research questions:

\textbf{RQ1—Impact.} Do obfuscations impact the energy usage of an application? If so, how?

\textbf{RQ2—Consistency.} Are there any significant differences in the impacts of the considered obfuscation tools or the considered obfuscation configurations?

\textbf{RQ3—Importance.} Are the impacts of applying obfuscations likely to be meaningful or noticeable to a typical mobile application user?

The remainder of this section discusses the results of our study in terms of these research questions.

A. RQ1: Impact

To gather the data necessary to answer our first research question, we performed Mann-Whitney-Wilcoxon (wilcox) tests to determine whether the difference between the amount of energy consumed by each scenario when run using the unobfuscated version of the application and each obfuscated version of the application is statistically significant. To check for statistical significance, we chose to use the Mann-Whitney-Wilcoxon test because we have one nominal variable (the obfuscation applied to the application), one measurement value (the amount of energy consumed by the execution), and we do not know whether our data are normally distributed. We chose an alpha (\(\alpha\)) of 0.05 and used R version 3.0.3’s implementation of the test (i.e., \texttt{wilcox.test}). Of the 270 tests that we conducted, 200 (\(\approx\) 74\%) indicated a statistically significant difference in the amount of energy consumed by the unobfuscated and obfuscated versions.

For the cases where there is a statistically significant difference (i.e., \(p \leq 0.05\)), we computed Vargha and Delaney’s \(A_{12}\) statistic to calculate the size of the effect of applying the obfuscation [5]. In general, the \(A_{12}\) statistic ranges from 0 to 1 and indicates, on average, how often one technique outperforms another: when \(A_{12}\) is exactly 0.5, the two techniques achieve equal performance; when \(A_{12}\) is less than 0.5, the first technique performs worse; and when \(A_{12}\) is greater than 0.5, the second technique is worse. The closer \(A_{12}\) is to 0 or 1, larger the effect. For our data, \(A_{12}\) represents the probability that the unobfuscated version consumes more energy than the obfuscated version.

Figure 2 shows the \(A_{12}\) statistics we calculated. In the figure, the y-axis shows the considered scenarios and the x-axis shows each obfuscation (combination of obfuscation tool and obfuscation configuration). For example, the first grouping shows the \(A_{12}\) statistics computed between the unobfuscated version of each application and the obfuscated versions produced by each obfuscation tool when using the all configuration. The color of each cell indicates the size and direction of the effect. Cells colored blue indicate cases where the unobfuscated version is more likely to consume more energy than the obfuscated version (i.e., \(A_{12} > 0.5\)) and cells that are colored red indicate cases where the unobfuscated version is more likely to consume less energy than the obfuscated version (i.e., \(A_{12} < 0.5\)). In addition, the color’s saturation indicates the size of the effect with the highest saturation indicating a “large” effect (\(A_{12}\) between 0.75 and 1.0 or between 0 and 0.25), a “medium” effect (\(A_{12}\) between 0.66 and 0.75 or between 0.25 and 0.33), or a “small” effect (\(A_{12}\) between 0.5 and 0.66 or between 0.33 and 0.5). Absent values indicate cases where there is not a statistically significant difference in energy usage between the versions.

From this data, we observe that, in general, obfuscations have a negative impact on energy usage (i.e., they increase energy usage). In 173 out of the 200 cases where there is a statistically significant difference in energy usage (\(\approx 87\%\) of the time), the obfuscated version is more likely to consume more energy than the unobfuscated version. In the remaining 27 cases (\(\approx 13\%\) of the time), the obfuscated version is more likely to consume less energy than the unobfuscated version. In addition, the size of the effect is most often “large”: the effect
size is “large” for 157 cases (≈79% of the time), “medium” for 34 cases (17% of the time), and “small” for 9 cases (≈4% of the time).

Next, we investigated the magnitude of the differences caused by the obfuscations. To determine the magnitude of the differences, we again focused on the cases where there is a significant difference in energy usage. For each combination of user scenario and obfuscation, we calculated the percentage change in mean of the energy usage between the obfuscated and the unobfuscated versions. The results of these computations are shown in Figure 3. The layout of this figure is similar to the layout of Figure 2. The y-axis shows the usage scenarios and the x-axis shows the obfuscations. The content of each cell shows the percentage change in mean energy usage. Again, the color of each cell indicates the direction and magnitude of the change. Blue cells indicate cases where the change is negative (i.e., energy usage decreased), red cells indicate cases where the change is positive (i.e., energy usage increased); darker colors indicate larger values, and absent values indicate cases where there is not a statistically significant difference in energy usage.

Overall, the percentage change in mean energy usage ranges from ≈−2% to ≈6.6% with a mean value of ≈1.7%. When considering only positive changes (i.e., increases in energy usage), the percentage change in mean energy usage ranges from 0.01% to ≈6.6% with a mean value of ≈2.1% and when considering only negative changes (i.e., decreases in energy usage), the percentage change in mean energy usage ranges from ≈−2% to ≈−0.1% with a mean value of ≈−0.6%. From these numbers, it is clear that not only is it more likely for an obfuscation to increase energy usage, the magnitude of negative impacts are larger than the magnitudes of positive impacts.

When compared to the energy impacts of other code level changes, the energy impacts of obfuscations are closer to the impacts of other focused changes (e.g., refactorings, whose impacts range from −7.5% to 4.5% [6] and switching application programming interface (API) implementations [7]) than to the impacts of more broad changes (e.g., applying design patterns, whose impacts can approach several hundred percent [8]).

Based on our investigations into the impacts of obfuscations on energy usage, we have found that:

1. Obfuscations can, and often do, impact the energy usage of an application with statistical significance.
2. Individually, all of our considered obfuscation tools and obfuscation configuration can both increase and decrease energy usage. However, there is one obfuscation (ZKM / se), that never caused a decrease in energy consumption.
3. The likelihood of causing an increase in energy usage is much higher than the likelihood of causing a decrease in energy usage.
4. The absolute values of the maximum and mean percentage changes for negative impacts are larger than the absolute values of the minimum and mean percentage changes for beneficial impacts.

B. RQ2: Consistency

The goal of our second research question is to determine whether there is a statistically significant benefit, with respect to energy usage, to using a specific obfuscation tool or specific obfuscation configuration. To answer this question, we performed several Kruskal-Wallis tests. We chose to use the Kruskal-Wallis test because we want to compare one measurement value (the amount of energy consumed by the execution) across compare multiple samples (obfuscation tools or obfuscation configurations) and we do not know whether our data are normally distributed. We chose an α of 0.05 and used R version 3.0.3’s implementation of the test (i.e., kruskal.test). In general, if the p value calculated by
Our first set of Kruskal-Wallis tests check whether there are any statistically significant differences in the percentage changes in mean energy usage among obfuscation tools for each obfuscation configuration. A $p$ value less than our chosen alpha would indicate that one of the obfuscation configurations is statistically different from the others. The results of these computations can be seen in Table IV. Because the computed $p$ values are never less than our chosen alpha ($0.05$), we cannot reject the null hypothesis. In practice, this means that, with respect to energy usage, there is no statistical benefit to picking one obfuscation tool over another. Consequently, developers are free to choose their preferred obfuscation tool based on other factors such as supported obfuscations, price, ease of use, etc., without having to worry about its impact on energy usage.

Our second set of Kruskal-Wallis tests check whether there are any statistically significant differences in the percentage changes in mean energy usage among the obfuscation configurations for each obfuscation tool. The result of these computations can be seen in Table V. Again, because the computed $p$ values are never less than our chosen alpha ($0.05$), we cannot reject the null hypothesis. In practice, this means that, with respect to energy usage, there is no statistical benefit to picking one obfuscation configuration over another. Again, application developers are free to choose their preferred obfuscation configurations based on factors other than its impact on energy usage.

While our initial belief about whether there is a significant difference among obfuscation tools matches the observed result, we were surprised that there is not a statistical difference among the obfuscation configurations. Initially, we believed that the opt configuration would improve energy usage and the rename configuration would have no effect on energy usage. While it is conceivable that the opt configuration makes changes that improve runtime performance at the expense of energy usage, it is less clear why changing the name of classes and methods often causes an increase in energy usage. After investigating this situation more closely, our current hypothesis is that renaming operations cause the more energy-expensive bytecodes to be used instead of the less expensive bytecodes. In future work, we plan on confirming on refuting this hypothesis with additional experimentation.

It is also interesting to note that the all configuration is not significantly different than the others. Because all

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>obfuscation configuration</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
</tr>
<tr>
<td>p value</td>
<td>0.53</td>
<td>0.53</td>
<td>0.53</td>
<td>0.53</td>
<td>0.53</td>
</tr>
</tbody>
</table>

Table IV. Kruskal-Wallis Results: % change ~ Obfuscation Tool.

The Kruskal-Wallis test is less than the chosen alpha, it indicates that at least one of the samples is significantly different from the others. It does not indicate how many differences occur or among which samples the differences exist. However, this information can be determined by running pairwise Mann-Whitney-Wilcoxon tests.
is a combination of the other configurations, we expected its impacts to be some function of the impacts of the other obfuscation configuration (e.g., sum or product). However, this is not the case. There is no statistical difference between all and the other configurations. Again, we plan to investigate the underlying reasons for this in future work.

C. RQ3: Importance

Our first research questions were primarily concerned with discovering if and how obfuscations impact the energy usage of applications. The goal of our third research question is to assess whether the observed impacts are likely to be meaningful or noticeable to mobile application users.

To answer this question, we first used Equation (1) to calculate the percentage of battery charge that is consumed by each scenario when it is executed using the unobfuscated version of its application and when it is executed using the obfuscated versions of its application.

\[
\%_{\text{charge}} = \frac{E}{3.8V} \times \frac{1000}{2100\, \text{mAh} \times 3600} \times 100
\]  

(1)

In Equation (1), \(E\) is the amount of energy in joules (J) consumed by an execution (here we used the mean energy consumption of each version of our 30 trials) and 3.8 V and 2,100 mAh are the voltage and electric charge of the Nexus 4’s battery, respectively. We then calculated, using Equation (2), the amount of time needed to drain the Nexus 4’s battery from full to empty (i.e., battery life) if the scenario were executed continuously using each version of its application.

\[
\text{t}_{\text{drain}} = \frac{100\%}{\%_{\text{charge}}} \times D
\]  

(2)

In Equation (2), \(\%_{\text{charge}}\) is the percentage of battery charge calculated using Equation (1) and \(D\) is the duration of the scenario (see Table I). Note that the unit of measurement for \(t_{\text{drain}}\) will be the same as the unit of measurement for \(D\). Finally, we computed the changes in battery life for each scenario and obfuscation tool/obfuscation configuration combination by subtracting the battery life of each obfuscated version from the battery life of the unobfuscated version.

Figure 4 shows the results of these computations. The left-most grouping, Bat. Life, shows the mean battery life in hours (h) when the unobfuscated version is run continuously, draining the battery from full to empty. For our applications, the mean battery life ranges from 3.0 h for Calculator and OpenSudoku with a mean of \(\approx 4.3\) h. The remaining five groupings show the change in mean battery life in minutes (min) when an obfuscated version is used instead of an unobfuscated version. Again, absent values indicate instances where there was no statistically significant difference in energy usage between the application versions and the color of each cell indicates the direction and magnitude of the change. Blue cells indicate obfuscations that increase battery life (i.e., changes that are beneficial for users) and red cells indicate obfuscations that decrease battery life (i.e., changes that are detrimental to users).

As the figure shows, across all configurations, (1) the change in battery life ranges from an increase of \(\approx 5\) min, when the View Lists scenario is run using the version of DailyMoney obfuscated using Allatori and the \(\text{opt}\) configuration, to a decrease of \(\approx 16\) min, when the View Lists scenario is run using the version of DailyMoney obfuscated using DashO and the \(\text{se}\) configuration and when the Add Detail scenario is run using the version of DailyMoney obfuscated using ZKM and the \(\text{opt}\) configuration, and (2) the mean change in battery life is \(\approx 4.3\) min. When only the \(\text{all}\) configuration for each obfuscation tool is considered, the results are similar. The change in battery life ranges from \(-14.9\) min to 2.8 min with a mean of \(\approx -3.8\) min.

Based on these results, we believe that it is unlikely for an application user to notice a decrease in battery life due to an obfuscation. Although we observed worse case decreases in battery life of \(\approx 6\) % (e.g., DailyMoney: Add Detail, DailyMoney: View Lists, and AnkiDroid: New Deck), recall that these are the expected changes if the scenarios were executed continually, draining the battery from full to empty. In practice this is unlikely since DailyMoney is a financial tracker.
that is designed to be used for short periods of time (e.g.,
recording a purchase) and AnkiDroid users are more likely
to review flash cards than create them. For the scenarios that
are much more likely to be run for long periods of time (e.g.,
OpenSudoku, FrozenBubblePlus, AnkiDroid: Tutorial Deck),
the change in battery life is typically less than 1%. Similarly,
we also believe that is unlikely for an application user to notice
an increase in battery life due to an obfuscation. Even in the
best case, the increase in battery life is \( \approx 2\% \).

In retrospect, this result makes sense. For mobile applica-
tions, the majority of energy is consumed by the phone’s
screen, radios, and sensors. The changes made by the obfus-
cations do not change how the applications interact with or
use these resources. Because the obfuscations make changes
to parts of the application that do not consume much energy,
the impacts of the obfuscations are overshadowed by the more
energy expensive parts of the execution.

While users are likely to be indifferent to this conclusion
because obfuscations neither harm nor improve their battery
life, it is good news for application developers. Now developers
are able to protect their applications by applying obfuscations
without needing to consider the obfuscation’s impacts on
energy usage.

D. Threats to Validity

One of the most significant threats to the validity of our
results is the possibility of energy usage measurement errors
either due to imprecise measurements or failing to control
for potential sources of noise. To minimize this threat, we
designed and implemented our EMP with the help of experi-
exenced electrical engineers from the University of Delaware’s
Department of Electrical and Computer Engineering. We also
took care to offload as much of the monitoring infrastructure
to external hardware so that it cannot add any overhead to the
energy usage of the execution that is being measured. Finally,
we used an existing capture / replay tool to deterministically
reproduce a recorded execution to eliminate biases in how the
applications are executed.

An additional threat to validity is that we considered 11
applications. Although we were careful in selecting these
applications, it is possible that they may not be representative
of all possible types of applications. Similarly, considering
only Android applications may prevent our conclusions from
generalizing to all environments where obfuscations are used
(e.g., Windows Phone applications, non-Android Java applica-
tions) and considering only a single device (i.e., the Nexus 4)
may prevent our conclusion from generalizing to other devices.
In future work, we plan to address these threats by expanding
our study to include additional applications and additional
devices.

A more specific concern is that, due to the limitations of
RERAN, our set of subject applications does not contain
applications that make heavy use of the network or sensors
beyond the touch screen. While network connections and
sensors are known to consume large amount of power, the
changes made by the obfuscation tools will not effect how the
applications use these resources (i.e., the number of types of
calls made to these APIs will not be changed). Consequently,
we believe that the results we observed will also hold for such
applications.

Finally, our choice of usage scenarios for driving each
application may not be representative of how the applications
are actually used in practice. However, we believe that this is
unlikely since the scenarios were generated from actual users
using each application for its intended purposes.

IV. RELATED WORK

The most closely related area of work is a recent group of
papers that have attempted to identify the underlying
causes of energy consumption by empirically investigating
the impact of various software development decisions. More
specifically, researchers have investigated the impacts of: refac-
torings [6], design patterns [8–10], sorting algorithms [11],
web servers [12], programming models [13, 14], lock-free
data structures [15], and API usage [7, 16]. In addition,
researchers have investigated trends in an application’s energy
consumption among versions [17] and how developers ask
questions about energy usage [18].

Another area of related work are techniques that attempt to
detect energy bugs (e.g., [19–22]). Although such techniques
are effective at detecting certain types of energy bugs, it is not
clear whether they would be able to detect unnecessary energy
usage as a result of applying obfuscations. In general these
techniques can only detect bugs that result in large, abnormal
spikes in energy usage or repeated increases in energy usage
without a corresponding user action (e.g., polling the GPS
when the screen is off). Applying an obfuscation is unlikely to
cause either of these situations to occur. Rather it is likely to
increase the overall energy usage by a small, constant amount.

Third, there is a significant amount of work focused on ac-
curately measuring energy consumption. Work in this area has
been conducted at various levels. Hardware instrumentation-
based approaches (e.g., [23]) use physical instrumentation to
measure the actual power usage of a system. Simulation-based
approaches (e.g., [24, 25]) use a cycle-accurate simulator to
replicate the actions of a processor at the architecture level and
estimate energy consumption of each executed cycle. Finally,
estimation-based approaches (e.g., [14, 26]) build models of
energy-influencing features and use such models to estimate
energy usage.

V. CONCLUSIONS AND FUTURE WORK

We have presented an empirical study that investigated the
impact of code obfuscations on the energy usage of mobile
applications. We considered 11 commonly used Android appli-
cations, 4 obfuscation tools, 5 obfuscation types, and 15 usage
scenarios. In total, we generated over 3.2 GB of experimental
data from 8,550 executions on our Nexus 4-based EMP. The
results of this study demonstrate that:

(1) Obfuscations can, and often do, impact the energy usage
    of applications with statistical significance.
(2) Obfuscations can both increase and decrease energy usage,
    but they are much more likely to increase energy usage.
(3) The magnitude of the impacts of obfuscations are compa-
    rable to the magnitude of the impacts of other code level
    changes such as applying refactorings.
(4) The differences between the impacts of the considered obfuscations on energy usage are not statistically significant.
(5) The impacts of obfuscation on battery life are unlikely to be meaningful to mobile application users.

As such, we believe that these results are positive news for both mobile application users and mobile application developers. Developers can protect their applications without impacting the battery life of the devices where their applications execute.

Based on these conclusions, there are several potential areas for future work. First, we plan on enlarging the scope of our study. Although we considered a significant number of applications and obfuscations, adding additional applications, platforms (e.g., tablets), and architectures (e.g., Windows phone) would potentially allow us to confirm or refute our observations. Second, we plan on investigating the underlying reasons for why the obfuscations exhibit the observed impacts and whether other types of information can be used to accurately predict the impacts of applying obfuscations. From the perspective of a software engineer, the answer to this question has a high utility. In most cases, developers do not have access to the type of custom, hardware-based energy profiling tools that we do. As a result, they have no way of identifying whether obfuscating their code has or will increase or decrease energy usage. Providing them with the ability to make accurate predictions about the impacts of applying obfuscations would be very useful.

VI. ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation Grant No. 1216488. We would also like to thank Smardec, PreEmptive Solutions, and Zelix Pty. Ltd. for providing full-featured copies of their obfuscation tools. Finally, we would like to thank Abram Hindle for his initial guidance on building the Nexus 4-based EMP.

REFERENCES