On the Energy Footprint of Mobile Testing Frameworks

Luís Cruz, Member, IEEE, and Rui Abreu, Senior Member, IEEE,

Abstract—High energy consumption is a challenging issue that an ever increasing number of mobile applications face today. However, energy consumption is being tested in an ad hoc way, despite being an important non-functional requirement of an application. Such limitation becomes particularly disconcerting during software testing: on the one hand, developers do not really know how to measure energy; on the other hand, there is no knowledge as to what is the energy overhead imposed by the testing framework. In this paper, as we evaluate eight popular mobile UI automation frameworks, we have discovered that there are automation frameworks that increase energy consumption up to roughly 2200%. While limited in the interactions one can do, Espresso is the most energy efficient framework. However, depending on the needs of the tester, Appium, Monkeyrunner, or UIAutomator are good alternatives. In practice, results show that deciding which is the most suitable framework is vital. We provide a decision tree to help developers make an educated decision on which framework suits best their testing needs.

Index Terms—Mobile Testing; Testing Frameworks; Energy Consumption.

1 INTRODUCTION

The popularity of mobile applications (also known as apps) has brought a unique, non-functional concern to the attention of developers – energy efficiency [1]. Mobile apps that (quickly) drain the battery of mobile devices are perceived as being of poor quality by users. As a consequence, users are likely to uninstall an app even if it provides useful functionality and there is no better alternative. In fact, mobile network operators recommend users to uninstall apps that are energy inefficient. It is therefore important to provide developers with tools and knowledge to ship energy efficient mobile apps [2], [3], [4], [5].

Automated testing tools help validate not only functional but also non-functional requirements such as scalability and usability [6], [7]. When it comes to energy testing, the most reliable approach to measure the energy consumption of mobile software is by using user interface (UI) automation frameworks [8], [9], [10], [11], [12], [13]. These frameworks are used to mimic user interaction in mobile apps while using an energy profiling tool. An alternative is to use manual testing but it creates bias, is error prone, and is both time and human resource consuming [14].

While using a UI automation framework is the most suitable option to test apps, there are still energy-related concerns that need to be addressed. By replicating interactions, frameworks are bypassing or creating overhead on system behavior. For instance, before executing a Tap, it is necessary to programmatically look up the target UI component. This creates extra processing that would not happen during an ordinary execution of the app. These idiosyncrasies are addressed in the work proposed in this paper, as they may have a negative impact on energy consumption results.

As a motivational example, consider the following scenario: an app provides a tweet feed that consists of a list of tweets including their media content (such as, pictures, GIFs, videos, URLs). The product owner noticed that users would tend to uninstall the app as its energy consumption was perceived as being too high. Therefore, the development team has to address this non-functional requirement.

One idea is to show plain text and pictures with low resolution. Original media content would be rendered upon a user Tap on the tweet, as depicted in Figure 1. With this approach, energy is potentially saved by rendering only media that the user is interested in. To validate this solution, developers created automated scripts to mimic user interaction in both versions of the app while measuring the energy consumption using a power meter. The script for the original version consisted in opening the app and scrolling the next 20 items, whereas the new version’s script consisted in opening the app and scrolling the next 20 items while tapping in 5 of them (a number they agreed to be the average hit rate of their users). A problem that arises is that the automation framework spends more energy to perform the five extra taps. Imagining that for each Tap the automation framework consumes 1 joule (J), the new version will have to spend at least 5 times less than the original version to be perceived as more efficient. If not, it gets rejected even though the new version could be more efficient.

More efficient frameworks could reduce this threshold to a more insignificant value. However, since automation frameworks have not considered energy consumption as an issue, developers do not have a sense of which framework is more suitable to perform reliable energy measurements.

1. Apigee’s survey on the reasons leading to bad mobile app reviews: https://cloud.google.com/apigee/news (visited on September 10, 2019)
3. Tap is a gesture in which a user taps the screen with his finger.
4. Joule (J) is the energy unit in the International System of Units.
The primary goal of this work is to study popular UI automation frameworks in the context of testing the energy efficiency of mobile apps. We address the following research questions:

- **RQ1**: Does the energy consumption overhead created by UI automation frameworks affect the results of the energy efficiency of mobile applications?
- **RQ2**: What is the most suitable framework to profile energy consumption?
- **RQ3**: Are there any best practices when it comes to creating automated scripts for energy efficiency tests?

We measure the energy consumption of common user interactions: Tap, Long Tap, Drag And Drop, Swipe, Pinch & Spread, Back button, Input text, Find by id, Find by description, and Find by content.

Results show that Espresso is the framework with best energy footprint, although Appium, Monkeyrunner, and UIAutomator are also good candidates. On the other side of the spectrum are AndroidViewClient and Calabash, which makes them not suitable to test the energy efficiency of apps yet. For a general purpose context, Appium follows as being the best candidate. We have further discovered that methods that use content to look up UI components need to be avoided since they are not energy savvy.

**Overheads incurred by UI automation frameworks ought to be considered when measuring energy consumption of mobile apps.**

To sum up, the main contributions of this paper are:

- A comprehensive study on energy consumption of user interactions mimicked by UI automation frameworks.
- Comparison of the state-of-the-art UI automation frameworks and their features in the context of energy tests.
- Best practices regarding the API usage of the framework for energy tests, including a decision tree to help choose the framework which suits one needs.

**2 RELATED WORK**

UI automation frameworks play an important role on the research of mobile software energy efficiency. They are used as part of the experimental setup for the validation of approaches for energy efficiency of mobile apps. Monkeyrunner has been used to assess the energy efficiency of Android’s API usage patterns [10]. It was found that UI manipulation tasks (e.g., method findViewByld) and database operations are expensive in terms of energy consumption. These findings suggest that UI automation frameworks might as well create a considerable overhead on energy consumption. Monkeyrunner has also been used to assess benefits in energy efficiency on the usage of Progressive Web Apps technology in mobile web apps [15], despite the fact that no statistically significant differences were found. Android View Client has been used to assess energy efficiency improvements of performance based optimizations for Android applications [16], [17], being able to improve energy consumption up to 5% in real, mature Android applications. Other works have also used Robotium [18], Calabash [12], [13], and RERAN [19], [20]. Our work uses a similar approach for assessing and validating energy efficiency, but it has distinct goals as we focus on the impact of UI automation frameworks on energy efficiency results.

Previous work studied five Android testing frameworks in terms of fragilities induced by maintainability [21], [22]. Five possible threats that could break tests were identified: 1) identifier change, 2) text change, 3) deletion or relocation of UI elements, 4) deprecated use of physical buttons, and 5) graphics change (mainly for image recognition testing techniques). These threats are aligned with efforts from existing works [23]. Our paper differentiates itself by focusing on the energy efficiency of Android testing tools.

In a study comparing Appium, MonkeyTalk, Ranorex, Robotium, and UIAutomator, Robotium and MonkeyTalk stood out as being the best frameworks for being easy to learn and providing a more efficient comparison output between expected and actual result [24]. A similar approach was taken in other works [25], [26] but although they provide useful insights about architecture and feature set, no systematic comparison was conducted. We compare different frameworks with a quantitative approach to prevent bias of results.

Linares-Vásquez M. et al. (2017) have studied the current state-of-the-art in terms of the frameworks, tools, and services available to aid developers in mobile testing [4]. It focused on 1) Automation APIs/Frameworks, 2) Record and Replay Tools, 3) Automated Test Input Generation Techniques, 4) Bug and Error Reporting/Monitoring Tools, 5) Mobile Testing Services, and 6) Device Streaming Tools. It envisions that automated testing tools mobile apps should address development restrictions: 1) restricted time/budget for testing, 2) needs for diverse types of testing (e.g., energy), and 3) pressure from users for continuous delivery. In a similar work, these issues were addressed by surveying 102 developers of Android open source projects [27]. This work identified a need for automatically generated test cases that can be easily maintained over time, low-overhead tools that can be integrated with the development workflow, and expressive test cases. Our work differs from these studies by...
providing an empirical comparison solely on UI automation frameworks, and addressing energy tests.

Choudhary R., et al. (2015) compared automated input generation (AIG) techniques using four metrics [28]: ease of use, ability to work on multiple platforms, code coverage, and ability to detect faults. It was found that random exploration strategies by Monkey\(^5\) or Dynodroid [29] were more effective than more sophisticated approaches. Although our work does not scope AIG tools, very often they use UI automation frameworks (e.g., UI/Automator and Robotium) underneath their systems [30], [31], [32], [33]. Results and insights about energy consumption in our study may also apply to tools that build on top of UI automation frameworks.

3 Design of the Empirical Study

To answer the research questions outlined in the Introduction, we designed an experimental setup to automatically measure energy consumption of Android apps. In particular, our methodology consists in the following steps:

1) Preparation of an Android device to use with a power monitor.
2) Creation of a stack of UI interaction scripts for all frameworks.
3) Automation of the execution of tests for each framework to run in batch mode.
4) Collection and analysis of data.

Our methodology is illustrated in Figure 2. There are three main components: a desktop computer that serves as controller; a power monitor; and a mobile device running Android, i.e., the device under test (DUT). The desktop computer sends interaction instructions to be executed in the mobile device. The power monitor collects energy consumption data from the mobile device and sends it to the desktop computer. Finally, the desktop computer analyzes data and generates reports back to the user.

3.1 Energy Data Collection

We have adopted a hardware-based approach to obtain energy measurements. We use Monsoon’s Original Power Monitor with the sample rate set to 5000Hz, as used in previous research [10], [11], [28], [34], [35], [36], [37], [38]. Measurements are obtained using the Physalia toolset\(^6\) – a Python library to collect energy consumption measurements in Android devices. It takes care of syncing the beginning and ending of the UI interaction script with the measurements collected from the power monitor. The steps described in Physalia’s tutorial\(^7\) were followed to remove the device’s battery and connect it directly to the Monsoon’s power source using a constant voltage of 3.8V. This is important to ascertain that we are collecting reliable energy consumption measurements.

3.2 Platform

The choice of the Android platform lies in the fact that it is one of the most popular operating systems (OS) and is open source. This helps to understand the underlying system and use a wide range of instrumentation tools. However, the techniques and ideas discussed in this paper apply to other operating systems as well.

3.3 UI Automation Frameworks

The state-of-the-art UI automation frameworks for Android used in our study are Appium, UI/Automator, PythonUI/Automator, AndroidViewClient, Espresso, Robotium, Monkeyrunner, and Calabash. The frameworks were chosen following a systematic criteria review: freely available to the community, open source, featuring a realistic set of interactions, expressed through a human readable and writable format (e.g., programming language), and used by the mobile development industry. To assess this last criterion StackOverflow and Github were used as proxy. Some frameworks have been discarded for not complying with this criteria. As an example, Ranorex\(^8\) is not free to the community and RERAN [19] is designed to be used with a recording mechanism. MonkeyTalk has not been publicly released after being acquired by Oracle\(^9\), and Selendroid is not ready to be used with the latest Android SDK\(^10\). We decided not to include UI recording tools since they rely on the underlying frameworks (e.g., Espresso Test Recorder, Robotium Recorder).

Although most frameworks support usage directly through screen coordinates, we only study the usage by targeting UI components. Usage through coordinates makes the tests cumbersome to build and maintain, and is not common practice.

An overview of the features of the frameworks is in Table 1. It also details the frameworks as to whether the app’s source code is required, whether it is remote script-based, i.e., simple interaction commands can be sent in real time to the DUT; WebView support, i.e., whether hybrid apps can also be automated; compatibility with iOS and supported programming languages. The most common languages supported by these frameworks are Python and Java.

---

5. UI/Application Exerciser Monkey also known as Monkey tool: https://developer.android.com/studio/test/monkey.html (visited on September 10, 2019).
We compare the overhead in energy consumption using as baseline the energy usage of interactions when executed by a human. Baselines for each interaction were measured by asking two Android users (one female and one male) to execute the interactions as in the automated scripts. For instance, in one of the experiments the participants had to click 200 times in the *Back Button*, we repeat 200 times; in the cases of *Swipe, Pinch and Spread*, or lookup methods, we repeat 40 times; in the remaining interactions, we repeat 10 times.

### 3.4 Test cases

For each framework, a script was created for every interaction that was supported by the framework, totaling 73 scripts. Scripts were manually and carefully crafted and peer reviewed to ascertain similar behavior across all frameworks. Essentially, each script calls a specific method of the framework that mimics the user interaction that we pretend to study. To minimize overheads from setup. The reason lies in the fact that Android automation frameworks resort to the Android Debug Bridge (ADB) to communicate with the mobile device. ADB allows to interact with the mobile device. ADB allows to enable Internet permission ("android.permission INTERNET").

### 3.5 Setup and Metrics

We compare the overhead in energy consumption using as baseline the energy usage of interactions when executed by a human. Baselines for each interaction were measured by asking two Android users (one female and one male) to execute the interactions as in the automated scripts. For instance, in one of the experiments the participants had to click 200 times in the *Back Button*. All interactions were measured except for *Find by id, Find by description, and Find by content*, as these are helper methods provided by the UI automation frameworks and are not applicable to human interactions.

As mentioned above, energy measurements are prone to random variations due to the nature of the underlying OS. Furthermore, one can also expect errors from the data collected from a power monitor [39]. To make sure energy consumption values are reliable and have enough data to perform significance tests, each experiment was identically and independently repeated 30 times.

Since user interactions often trigger other tasks in a mobile device, tests have to run in a controlled environment. In other words, we are trying to measure the platform overhead and we don't want the app activity to interfere with that measurement. Thus, an Android application was developed by the authors for this particular study. It differs from a real app in the sense that this app is a strategy to prevent any extra work from being performed by the foreground activity. The main goal is preventing any side-effect from UI interactions, which in real apps would result in different behaviors, hence compromising measurements. Hence, the app prevents the propagation of the system’s event created by the interaction and no feedback is provided to the user. This way, experiments only measure the work entailed by frameworks.

The main settings used in the device are listed in Table 2. Android provides system settings that can be useful to control the system behavior during experiments. Notifications and alarms were turned off, lock screen security was disabled, and the “Don’t keep Activities” setting was enabled. This last setting destroys every activity as soon as the user leaves it, erasing the current state of an app. 11

WiFi is kept on as a requirement of our experimental setup. The reason lies in the fact that Android automation frameworks resort to the Android Debug Bridge (ADB) to communicate with the mobile device. ADB allows to...
stall/uninstall/open apps, send test data, configure settings, lock/unlock the device, among other things. By default, it works through USB, which interferes with energy consumption measurements. Although Android provides settings to disable USB charging, if the USB port remains connected to the device, the measurements of energy consumption become obsolete. Fortunately, ADB can be configured to be used through a WiFi connection, which was leveraged in this work.

In addition to the energy consumption sources mentioned before, there is another common one – the cost of having the device in idle mode. In this context, we consider idle mode when the device is active with the settings in Table 2 but is not executing any task. In this mode, the screen is still consuming energy. We calculate the idle cost for each experiment to assess the effective energy consumption of executing a given interaction. We measure the idle cost by collecting the energy usage of running the app for 120 seconds without any interaction. In addition to the mean energy consumption, we compare different frameworks using the mean energy consumption without the corresponding idle cost, calculated as follows:

$$x' = \frac{\sum_{i=1}^{N=30} (E_i - IdleCost \times \Delta t_i)}{N}$$

(1)

where $N$ is the number of times experiments are repeated (30), $E_i$ is the measured energy consumption for execution $i$, $IdleCost$ is the energy usage per second (i.e., power) of having the device in idle mode, expressed in watts (W), and $\Delta t_i$ the duration of execution $i$.

After removing idle cost, we compute overhead in a similar fashion as previous work [40]:

$$Overhead(\%) = \left(\frac{x' \times \text{human}}{x' \times \text{human} - 1}\right) \times 100$$

(2)

In other words, overhead is the percentage change of the energy consumption of a framework when compared to the real energy consumption induced by human interaction.

We also use $x'$ to compute the estimated energy consumption for a single interaction ($S_g$) as follows:

$$S_g = x' / M$$

(3)

where $M$ is the number of times the interaction was repeated within the same execution (e.g., in Back Button, $M = 200$).

Experiments were executed using an Apple iMac Mid 2011 with a 2.7GHz Intel Core i5 processor, 8GB DDR3 RAM, and running OS X version 10.11.6. Room temperature was controlled for 24°C (75°F). DUT was a Nexus 5X manufactured by LG, running Android version 6.0.1. All scripts, mobile app, and data are available in the Github repository of the project\footnote{Project’s Github repository: https://github.com/luiscruz/physalia-automators visited on September 10, 2019.}, which is released with an open source license.

4 Results

Next, we report the results obtained in the empirical study.

4.1 Idle Cost

In a sample of 30 executions, the mean energy consumption of having the app open for 120 seconds without any interaction is 22.67J. The distribution of the measurements across the 30 executions is shown in Figure 3. This translates into a power consumption of 0.19W (in other words, the app consumes 0.19 joules per second in idle mode). This value is used in the remaining experiments to factor out idle cost from the results.

4.2 Tap

Table 3 presents results for the Tap interaction. Each row in the table describes a framework as a function of the mean energy consumption ($\bar{x}$), standard deviation of energy consumption ($s$), duration of each execution of the script ($\Delta t$) in seconds, the mean energy consumption without idle cost ($x'$, see Eq. 1), the estimated energy consumption for a single interaction ($S_g$, see Eq. 3), the Cohen’s-$d$ effect size ($d$), the percentage overhead when compared to the same interaction when executed by a human (as in Eq. 2), and the position in the ranking (#), i.e., the ordinal position when results are sorted by the average energy consumption, and . With the exception of the results for Human which are placed in the first row, the table is sorted in alphabetical order for the sake of comparison with results of other interactions.

From our experiments, we conclude that Espresso is the most energy efficient framework for Taps, consuming 3.63J on average after removing idle cost, while a single Tap is estimated to consume 0.09J. When compared to the human interaction, Espresso imposes an overhead of 16%. The least efficient frameworks for a Tap are Appium, and Calabash, with overheads of 871% and 470%, respectively. Using these frameworks for taps can dramatically affect energy consumption results.

A visualization of these results is in Figure 4. The height of each white bar shows the mean energy consumption for the framework. The height of each green or yellow bar represents the energy consumption without the idle cost. The yellow bar and the dashed horizontal line highlight the baseline energy consumption. In addition, it shows a violin plot with the probability density of data using rotated kernel density plots. The violin plots provide a visualization of the distribution, allowing to compare results regarding shape, location, and scale. This is useful to assess whether data can be modeled with a normal distribution, and compare
the standard deviations of the measurements in different frameworks.

### 4.3 Long Tap

Results for the interaction Long Tap are in Table 4 and Figure 5. Monkeyrunner and Espresso are the most efficient frameworks, with overheads of 77% ($\bar{x}' = 12.60$) and 81% ($\bar{x}' = 12.88$), respectively. PythonUIAutomator and Calabash are the most inefficient (overhead over 300%).

A remarkable observation is the efficiency of Appium’s Long Tap ($S_g = 0.40$) when compared to its regular Tap ($S_g = 0.76$). Common sense would let us expect Tap to spend less energy than Long Tap, but that is not the case. This happens because Appium’s usage of Long Tap requires a manual instantiation of a TouchAction object, while Tap creates it internally. Although creating such object makes code less readable, the advantage is that it can be reused for the following interactions, making a more efficient use of resources.

### 4.4 Drag and Drop

Results for the interaction Drag and Drop in Table 5 and Figure 6. UIAutomator is the best testing framework with an overhead of 185% ($x' = 14.48$). Espresso is not included in the experiments since Drag and Drops are not supported. The most energy greedy framework is Calabash with an overhead of 2193%. When compared to UIAutomator, one Drag and Drop with Calabash is equivalent to more than 11 Drag and Drops. Hence, Calabash should be avoided for energy measurements that include Drag and Drops.

### 4.5 Swipe

Results for the interaction Swipe are presented in Table 6 and Figure 7. Espresso is the best framework with an overhead of 29%, while Robotium, AndroidViewClient, Monkeyrunner, and Calabash are the most energy greedy with similar overheads, above 400%.

### 4.6 Pinch and Spread

Results for the interaction Pinch and Spread are presented in Table 7 and Figure 8. Although this interaction is widely used in mobile applications for features such as zoom in and out, only Calabash, PythonUIAutomator, and UIAutomator support it out of the box. UIAutomator is the most efficient framework, spending less energy than the equivalent interaction performed by a human (~5%). The remaining frameworks, PythonUIAutomator and Calabash were not as efficient, providing overheads of 181% and 374%, respectively.

### Table 4

Descriptive statistics of Long Tap interaction.

<table>
<thead>
<tr>
<th>Framework</th>
<th>$\bar{x}$</th>
<th>$\bar{s}$</th>
<th>$\bar{S_g}$</th>
<th>d</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>10.33</td>
<td>2.21</td>
<td>32.86</td>
<td>7.12</td>
<td>177.92</td>
</tr>
<tr>
<td>AndroidViewClient</td>
<td>49.18</td>
<td>5.24</td>
<td>119.21</td>
<td>26.66</td>
<td>666.40</td>
</tr>
<tr>
<td>Appium</td>
<td>25.34</td>
<td>0.86</td>
<td>49.60</td>
<td>15.96</td>
<td>399.08</td>
</tr>
<tr>
<td>Calabash</td>
<td>46.96</td>
<td>1.81</td>
<td>94.27</td>
<td>29.14</td>
<td>728.37</td>
</tr>
<tr>
<td>Espresso</td>
<td>19.87</td>
<td>0.54</td>
<td>37.00</td>
<td>12.88</td>
<td>327.02</td>
</tr>
<tr>
<td>Monkeyrunner</td>
<td>21.68</td>
<td>0.74</td>
<td>48.07</td>
<td>12.60</td>
<td>315.04</td>
</tr>
<tr>
<td>PythonUIAutomator</td>
<td>48.19</td>
<td>12.63</td>
<td>101.13</td>
<td>29.08</td>
<td>727.02</td>
</tr>
<tr>
<td>Robotium</td>
<td>39.35</td>
<td>1.82</td>
<td>99.97</td>
<td>20.46</td>
<td>511.40</td>
</tr>
<tr>
<td>UIAutomator</td>
<td>22.39</td>
<td>0.75</td>
<td>45.40</td>
<td>13.81</td>
<td>345.20</td>
</tr>
</tbody>
</table>

### Table 5

Descriptive statistics of Drag and Drop interaction.

<table>
<thead>
<tr>
<th>Framework</th>
<th>$\bar{x}$</th>
<th>$\bar{s}$</th>
<th>$\bar{S_g}$</th>
<th>d</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>14.48</td>
<td>0.63</td>
<td>30.27</td>
<td>8.76</td>
<td>219.02</td>
</tr>
<tr>
<td>AndroidViewClient</td>
<td>14.48</td>
<td>0.63</td>
<td>30.27</td>
<td>8.76</td>
<td>219.02</td>
</tr>
<tr>
<td>Appium</td>
<td>20.63</td>
<td>1.02</td>
<td>52.17</td>
<td>10.77</td>
<td>269.29</td>
</tr>
<tr>
<td>Calabash</td>
<td>36.30</td>
<td>3.77</td>
<td>93.53</td>
<td>18.62</td>
<td>465.56</td>
</tr>
<tr>
<td>Monkeyrunner</td>
<td>28.50</td>
<td>1.29</td>
<td>52.97</td>
<td>18.49</td>
<td>462.22</td>
</tr>
<tr>
<td>PythonUIAutomator</td>
<td>41.30</td>
<td>0.67</td>
<td>96.00</td>
<td>23.16</td>
<td>295.69</td>
</tr>
<tr>
<td>Robotium</td>
<td>43.27</td>
<td>0.81</td>
<td>93.73</td>
<td>25.56</td>
<td>319.46</td>
</tr>
<tr>
<td>UIAutomator</td>
<td>27.56</td>
<td>0.65</td>
<td>60.13</td>
<td>16.20</td>
<td>202.49</td>
</tr>
</tbody>
</table>

### Table 6

Descriptive statistics of Swipe interaction.

<table>
<thead>
<tr>
<th>Framework</th>
<th>$\bar{x}$</th>
<th>$\bar{s}$</th>
<th>$\bar{S_g}$</th>
<th>d</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>9.11</td>
<td>3.09</td>
<td>24.49</td>
<td>4.48</td>
<td>56.00</td>
</tr>
<tr>
<td>AndroidViewClient</td>
<td>45.29</td>
<td>0.62</td>
<td>115.87</td>
<td>23.39</td>
<td>292.41</td>
</tr>
<tr>
<td>Appium</td>
<td>17.09</td>
<td>0.46</td>
<td>30.00</td>
<td>11.42</td>
<td>142.80</td>
</tr>
<tr>
<td>Calabash</td>
<td>43.27</td>
<td>0.81</td>
<td>93.73</td>
<td>25.56</td>
<td>319.46</td>
</tr>
<tr>
<td>Espresso</td>
<td>10.35</td>
<td>0.26</td>
<td>24.10</td>
<td>5.79</td>
<td>72.43</td>
</tr>
<tr>
<td>Monkeyrunner</td>
<td>36.63</td>
<td>1.49</td>
<td>68.67</td>
<td>23.65</td>
<td>295.69</td>
</tr>
<tr>
<td>PythonUIAutomator</td>
<td>26.42</td>
<td>0.91</td>
<td>74.60</td>
<td>16.11</td>
<td>201.32</td>
</tr>
<tr>
<td>Robotium</td>
<td>41.30</td>
<td>0.67</td>
<td>96.00</td>
<td>23.16</td>
<td>289.46</td>
</tr>
<tr>
<td>UIAutomator</td>
<td>27.56</td>
<td>0.65</td>
<td>60.13</td>
<td>16.20</td>
<td>202.49</td>
</tr>
</tbody>
</table>
4.7 Back Button

Results for the interaction Back Button are presented in Table 8 and Figure 9. In this case, human interaction was considerably less efficient than most frameworks, being ranked fifth on the list. The main reason for this is that frameworks do not realistically mimic the Back Button interaction. When the user presses the back button, the system produces an input event and a vibration or haptic feedback simultaneously. However, frameworks simply produce the event. Thus, results are not comparable with the human interaction. Still, AndroidViewClient provided an overhead of 440%, being the most inefficient framework.

Another remarkable result was that Robotium, despite being energy efficient after removing idle cost, is the slowest framework. Thus, it is likely that Robotium is using a conservative approach to generate events in the device: it suspends the execution to wait for the back button event to take effect in the app.

4.8 Input Text

Results for the interaction Input Text are presented in Table 9 and Figure 10. Each iteration of Input Text consists in writing a 17-character sentence in a text field and then clearing it all back to the initial state. Thus, the value for a single interaction (\(S_g\)) is the energy spent when this sequence of events is executed, but can hardly be extrapolated for other input interactions.

\(\text{UIAutomator}\) is the framework with the lowest energy consumption (\(x_s = 1.42\)). The human interaction spends more energy than most frameworks. The reason behind this is that frameworks have a different way to deal with text input. Most frameworks generate a sequence of events that will generate the given sequence of characters. On the contrary, the human interaction resorts to the system keyboard to generate this sequence. Thus the system has to process a sequence of taps and match it to the right character event. There are even other frameworks, namely \(\text{UIAutomator}, \text{PythonUIAutomator},\) and \(\text{Robotium},\) that, as showed in the overview of Table 1, implement Input Text more artificially. Instead of generating the sequence of events, they directly change the content of the text field. This is more efficient but bypasses system and application behavior – e.g., automatic text correction features.

Results showed that the AndroidViewClient is very inefficient and its overhead (936%) is not negligible when measuring the energy consumption of mobile apps.

4.9 Find by id

Results for the task Find by id are presented in Table 10 and Figure 11. Find by id is a method that looks up for a UI component that has the given id. It does not mimic any user...
interaction but it is necessary to create interaction scripts. Methods Find by description and Find by content are used to achieve the same objective. For this reason, we do not report the consumption of a human interaction in these cases.

For the sake of consistency with previous cases, we report tables and figures in the same fashion. However, we consider that the overall cost of energy consumption (without removing idle cost) should not be discarded.

Robotium is the most energy efficient, with an energy consumption without idle cost of 0.94 J. However, if we consider idle cost, Robotium is amongst the most energy greedy frameworks (after Calabash and AndroidViewClient). It has an overall energy consumption of 27.97 J. When considering idle cost, Espresso is the most energy efficient framework.

This difference lies in the mechanism adopted by frameworks to deal with UI changes. After user interaction, the UI is expected to change and the status of the UI can become obsolete. Thus, frameworks need to wait until the changes the UI are complete. Results show that Robotium uses a mechanism based on suspending the execution to make sure the UI is up to date. On the other hand, Espresso uses a different heuristic, which despite spending more energy on computation tasks, it does not require the device to spend energy while waiting.

### 4.10 Find by description

Results for Find by description are presented in Table 11 and Figure 12. Find by description and Find by id are very similar regarding usage and implementation, which is confirmed by results. Espresso is the best framework regardless of idle cost ($\bar{x} = 1.37$ J and $\bar{x'} = 0.97$ J). Android View Client and Calabash are distinctly inefficient. All other frameworks show reasonable energy footprints, except for Robotium and Monkeyrunner, which were not included since Find by description is not supported.

### 4.11 Find by content

Results for Find by content are presented in Table 12 and Figure 13. After removing idle cost, Robotium is the framework with best results ($\bar{x'} = 0.14$ J). However, in resemblance to Find by id, Robotium is very inefficient when idle cost is not factored out ($\bar{x} = 23.74$ J). In this case, Appium is the most efficient framework ($\bar{x} = 3.07$ J).

Unlike with Find by id and Find by description, Espresso did not yield good results in this case ($\bar{x} = 9.43$ J and $\bar{x'} = 6.19$ J). This is explained by the fact that Espresso runs natively on the DUT. Thus, finding a UI component by content requires extra processing: the DUT has to search for a pattern on the DUT. Thus, finding a UI component by content requires extra processing: the DUT has to search for a pattern on the DUT. Thus, finding a UI component by content requires extra processing: the DUT has to search for a pattern on the DUT.

### 4.12 Statistical significance

As expected from previous work and corroborated with the violin plots, our measurements follow a normal distribution.
– also confirmed with the Shapiro-Wilk test. Thus, we assess Welch’s t-test as used in previous work [16]. We apply the Benjamini-Hochberg procedure by correcting p-values with the number of times a given sample is used in different tests.

All but a few tests (2 out 105) resulted in a small p-value, below the significance level $\alpha = 0.05$. For those pairs where there was no statistical significance, we could not find any meaningful finding. Given the myriad number of tests performed, results are not presented. Violin plots corroborate statistical significance by presenting very distinct distributions among all different frameworks. For further details, all results and data are publicly available\(^\text{13}\).

### 4.13 Threats to validity

Construct validity: Frameworks rely on different approaches to collect information about the UI components that are visible on the screen. The app used in the experiments has a UI that remains unchanged upon user interactions. In a real scenario, however, the UI typically reacts to user interactions. Frameworks that have an inefficient way of updating their UI model of components visible in the screen, may entail a high overhead on energy consumption. However, as manually triggering this update is not supported in most frameworks, it was unfeasible to include it in our study.

In addition, the overheads are calculated based on the results collected from the human interaction from two participants. Although results showed a small variance between different participants, the energy consumption may vary with other humans. Nevertheless, differences are not expected to be significant, and results still apply.

Moreover, energy consumption for a single interaction is inferred by the total consumption of a sequence of interactions. Potential tail energy consumptions\(^\text{14}\) of a single interaction are not being measured. This is mitigated by running multiple times the same interaction.

Internal validity: The Android OS is continuously running parallel tasks that affect energy consumption. For that reason, system settings were customized as described in Section 3 (e.g., disabled automated brightness and notifications). Also, each experiment is executed 30 times to ensure statistical significance as recommended in related work [10].

UI interactions typically trigger internal tasks in the mobile application running in foreground. The mobile application used in experiments was developed to prevent any side-effects to UI events. To ensure that scripts are interacting with the device as expected, the application was set to a mode that is not affected by user interaction. Thus, the behavior is equal across different UI automation frameworks and experiments only measure their energy consumption.

Finally, our experiments use a WiFi-configured ADB instead of a USB connection. This is a requirement from remote script-based frameworks. We did not measure the energy consumption entailed from using a USB-configured ADB. Nevertheless, we do not expect results to differ since the WiFi connection is only used before and after the measurements.

External validity: Energy consumption results vary upon different versions of Android OS, different device models, and different framework version. However, unless major changes are released, results are not expected to significantly deviate from the reported ones. Note that testing different devices requires disassembling them and making them useless for other purposes (that is to say that empirical studies as the one conducted by us are expensive), which can be economically unfeasible. Regardless, all the source code used in experiments will be released as Open Source to foster reproducibility.

### 5 Discussion

By answering the research questions, in this section we discuss our findings from the empirical evaluation, as well as outline their practical implications.

**RQ1: Does the energy consumption overhead created by UI automation frameworks affect the results of the energy efficiency of mobile applications?**

Yes, results show that interactions can have a tremendous overhead on energy consumption when an inefficient UI automation framework is used.

According to previous work, executing a real app during 100s yields an energy consumption of 58 J, on average [8]. Considering our results, executing a single interaction such as Drag and Drop can increase energy consumption in 1.7 times (overhead of 3% in this case). However, given that mobile apps are very reactive to user input [41], in 100 seconds of execution, more interactions are expected to affect energy.
Although a fair comparison must control for different devices and OS versions, this order of magnitude implies that overheads are not negligible. Thus, choosing an efficient UI automation framework is quintessential for energy tests.

Since all frameworks produce the same effect in the UI, the overhead of energy consumption is created by implementation decisions of the framework and not by the interaction itself. The main goal of a UI testing framework is to mimic realistic usage scenarios, but interactions with such overhead can be considered unrealistic.

One practical implication of the results in this work is to drive a change in the mindset of tool developers, bringing awareness of the energy consumption of their frameworks. Thus, we expect future releases of UI automation frameworks to become more energy efficient.

AndroidViewClient and Calabash consistently showed poor energy efficiency among all interactions. Despite providing a useful and complete toolset for mobile software developers, they should be used with prudence while testing the energy consumption of an app that heavily relies on user interactions. Work of Carette A. et al. (2017) was affected by a poor choice of framework: the authors used Calabash to mimic between 136 and 325 user interactions in experiments that, in total, consume roughly 350J. Considering our results, a single tap with Calabash is equivalent to 0.45J – it means that at least 60J (17%) were spent by the UI framework. The same interactions with Espresso would have been reduced to 12J (3%). The impact increases when considering other interactions. Our work shows that results would be different if the overhead of the framework had been factored out. On the contrary, Calabash was also used in other work [12] but its impact can be considered insignificant since experiments did not require much interaction and the main source of energy consumption came from Web page loads. Note, however, that the measurement setup is different and results from related work are not directly comparable. We plan to address this analysis in future work. In any case, we consider that using a more energy efficient framework could corroborate the evidence or find new – even contradictory – conclusions.

RQ2: What is the most suitable framework to profile energy consumption?

Choosing the right framework for a project can be challenging: there is no one solution fits all. Based on our observations, Figure 14 depicts a decision tree to help software developers making an educated guess about the most suited and energy efficient framework, given the idiosyncrasies of an app (that may restrict the usage of a framework). For example, if the project to be tested requires WebView support, one should use Appium rather than the other frameworks. Robotium is also an option if the app requires Taps or Input Text only, and neither iOS support nor remote scripting is required.

Remote script-based frameworks allow developers to easily create automation scripts. The script can be iteratively created using a console while interactions take effect on the phone in real time. From our experience while doing this work, remote script-based frameworks are easier to use and set up (i.e., gradual learning curve). This is one of the reasons many frameworks decided to use scripting languages (e.g., Python and Ruby) instead of the official languages for Android, Java or Kotlin. Notwithstanding, remote script-based frameworks require an active connection with the phone during measurements, which leads to higher energy consumption (as is confirmed by results). Each step of the interaction requires communication with the DUT; hence, the communication logic unavoidably increases the energy consumption. On the contrary, other frameworks can transfer the interaction script in advance to the mobile phone and run it natively on the phone, which is more energy efficient.

There are, however, two scenarios where remote script-based frameworks exhibit the best results: Back Button with Monkeyrunner (see Table 8), and Find By Content with Appium (see Table 12). This is an interesting finding as it shows that remote script-based frameworks can also be developed in an energy efficient way. As such, this evidence shows that there is room for energy optimization in the other frameworks.

In addition, USB communication is out of question for remote script-based frameworks since it affects the reliability of measurements. Frameworks that do not support remote scripting can be used with USB connection if unplugged during measurements (using tools such as Monsoon Power Monitor).

Among remote script-based frameworks, Monkeyrunner is the most energy efficient framework. The only problem is that it does not support many of the studied interactions. These results show that if energy consumption turns into a priority, it is possible to make complex frameworks such as Appium more energy efficient.

There are a number of other fine-grained requirements that developers need to consider when choosing a UI framework. A more thorough decision ought to consider other factors, such as existing infrastructure, development process, and learning curve. Nevertheless, we argue that decision tree of Figure 14 provides an approximate insight even though it does not take all factors into account.

RQ3: Are there any best practices when it comes to creating automated scripts for energy efficiency tests?

One thing that stands out is the fact that looking up one UI component is expensive. This task is exclusively required for automation and does not reflect any real-world interaction. Taking the example of Espresso: a single Tap consumes 0.09J, while using content to look up a component consumes 0.05J. Since a common Tap interaction requires looking a component up, 36% of energy spent is on that task.

Looking up UI components is energy greedy because the framework needs to process the UI hierarchy find a component that matches a given id, description, or content. Since the app we use has a very simple UI hierarchy, the energy consumption is likely to be higher in real apps. Hence, using lookup methods should be avoided whenever possible. A naive solution could be using the pixel position of UI components instead of identifiers. Pixel positions could be collected using a recorder. However, this is a bad practice since it brings majors maintainability issues across different releases and device models. For that reason, state-of-the-art UI recorders used by Android developers, such as
Robotium Recorder, yield scripts based on UI identifiers. As an alternative, we recommend caching the results of lookup calls whenever possible.

In addition, lookup methods Find by Id and Find by Description should be preferred to Find by Content. Results consistently show worse energy efficiency when using Find by Content. In Espresso, this difference gives an increase in energy consumption from 1.4\(\text{J}\) to 9.4\(\text{J}\) (overhead of 600%).

6 Conclusion

In this paper, we analyze eight popular UI automation frameworks for mobile apps with respect to their energy footprint. UI interactions have distinct energy consumptions depending on the framework. Our results show that the energy consumption of UI automation frameworks should be factored out to avoid affecting results of energy tests. As an example, we have observed the overhead of the Drag and Drop interaction to go up to 2200\%. Thus, practitioners and researchers should opt for energy efficient frameworks. Alternatively, the energy entailed by automated interactions must be factored out from measurements.

Espresso is observed to be the most energy efficient framework. Nevertheless, it has requirements that may not apply to all projects: 1) requires access to the source code, 2) does not support complex interactions such as Drag and Drop and Pinch and Spread, 3) is not compatible with WebViews, 4) is OS dependent, and 5) is not remote script-based. Hence, there are situations where UIAutomator, Monkeyrunner, and Appium are also worth considering. For a more general purpose context, Appium follows as being the best candidate. Thus, we propose a decision tree (See Figure 14) to help in the decision-making process.

Furthermore, we have also noticed the following in our experiments. Helper methods to find components in the interface are necessary when building energy tests, but should be minimized to prevent affecting energy results. In particular, lookup methods based on the content of the UI component need to be avoided. They consistently yield poor energy efficiency when compared to lookups based on id (e.g., in Espresso it creates an overhead of 600%).

This work paves the way for the inclusion of energy tests in the development stack of apps. It brings awareness to the energy footprint of tools used for energy test instrumentation, affecting both academic and industrial use cases. It remains to future work to design a catalog of energy-aware testing patterns.15

Acknowledgments

This work was partially funded by FCT - Fundação para a Ciência e a Tecnologia with reference UID/EIA/50014/2019, the GreenLab Project (ref. POCI-01-0145-FEDER-016718), the FaultLocker Project (ref. PTDC/CCI-COM/29300/2017), and the Delft Data Science (DDS) project.

References


15 For instance, augmenting/improve the following list of testing patterns http://wiki.c2.com/?TestingPatterns

Fig. 14. Selecting the most suitable framework for energy measurements.


