On the Energy Footprint of Mobile Testing Frameworks

Article *in* IEEE Transactions on Software Engineering · October 2019

DOI: 10.1109/TSE.2019.2946163

2 authors, including:

Luís Cruz
Delft University of Technology
25 PUBLICATIONS **217** CITATIONS

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[1]. 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[2]. 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*[3], 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 rather value apps with low energy consumption. Hence, 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 scroll 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[4] (J), the new version will have to spend at least 5J 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.

- *Luís Cruz is with Delft University of Technology. E-mail: see https://luiscruz.github.io/*
- *Rui Abreu is with University of Lisbon and INESC-ID. E-mail: see https://ruimaranhao.com/*

Manuscript received April 19, 2005; revised August 26, 2015.

1. *Apigee*'s survey on the reasons leading to bad mobile app reviews: https://cloud.google.com/apigee/news (visited on September 10, 2019)
2. *High Risk Android Apps*: https://www.verizonwireless.com/support/services-and-apps/ (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.

a) Original b) Refactored

Fig. 1. Two versions of the example app.

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 `findViewById`) 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

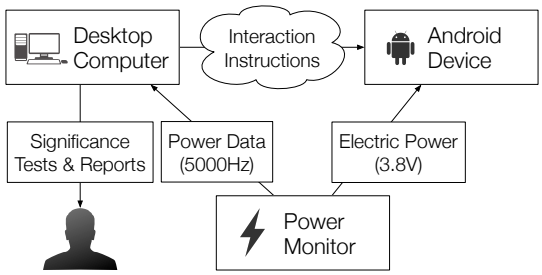

Fig. 2. Experimentation system to compare UI automation frameworks for Android.

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., *UIAutomator* 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*, *UIAutomator*, *PythonUIAutomator*, *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).

6. Physalia's webpage: https://tqrg.github.io/physalia/ (visited on September 10, 2019).

7. Tutorial's webpage: https://tqrg.github.io/physalia/monsoon_tutorial (visited on September 10, 2019).

8. *Ranorex*'s website available at https://www.ranorex.com (visited on September 10, 2019).

9. More information about *MonkeyTalk*'s acquisition: https://www.oracle.com/corporate/acquisitions/cloudmonkey/ (visited on September 10, 2019)

10. Running Selendroid would require changing its source code: https://github.com/selendroid/selendroid/issues/1116 and https://github.com/selendroid/selendroid/issues/1107 (visited on September 10, 2019)

TABLE 1
Overview of the studied UI automation frameworks

Framework	Android View Client	Appium	Calabash	Espresso	Monkeyrunner	Python Ui Automator	Robotium	UIAutomator
Tap	✓	✓	✓	✓	✓	✓	✓	✓
Long Tap	✓	✓	✓	✓	✓	✓	✓	✓
Drag And Drop	✓	✓	✓	×	✓	✓	✓	✓
Swipe	✓	✓	✓	✓	✓	✓	✓	✓
Pinch & Spread	×	×	✓	×	×	✓	×	✓
Back button	✓	✓	✓	✓	✓	✓	✓	✓
Input text	✓	✓	✓	✓	✓	✓(*)	✓(*)	✓(*)
Find by id	✓	✓	✓	✓	✓	✓	✓	✓
Find by description	✓	✓	✓	✓	×	✓	×	✓
Find by content	✓	✓	✓	✓	×	✓	✓	✓
Tested Version	13.2.2	1.6.3	0.9.0	2.2.2	n.a.	0.3.2	5.6.3	2.1.2
Min Android SDK	All	Recmd. $\geq$ 17	All	$\geq$ 8	n.a.	$\geq$ 18	$\geq$ 8	$\geq$ 18
Black Box	Yes	Yes	Limited (**)	No	Yes	Yes	Yes	Yes
Remote script-based	Yes	Yes	Yes	No	Yes	Yes	No	No
WebView Support	Limited	Yes	Yes	Yes	Limited	Limited	Yes	Limited
iOS compatible	No	Yes	Yes	No	No	No	No	No
BDD support	No	Yes	Yes	Yes	No	No	Yes	Limited
Integration test	Yes	Yes	Yes	No	No	Yes	Yes	Yes
Language	Python	Any WebDriver compatible lang.	Gherkin/Ruby	Java	Jython	Python	Java	Java
License	Apache 2.0	Apache 2.0	EPL 1.0	Apache 2.0	Apache 2.0	MIT	Apache 2.0	Apache 2.0
SOverflow Qns	164	3,147	569	292	437	0	1,012	438
Github Stars	540	5,514	1,429	n.a.	n.a.	719	2,165	n.a.

(*) Although it supports *Input Text*, it does not apply a sequential input of key events. This is more energy efficient but it is more artificial, bypassing real behavior (e.g., auto correct).
(**) Requires to manually enable Internet permission ("`android.permission.INTERNET`").

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 tasks (e.g., opening the app, getting app's UI hierarchy), the method is repeated multiple times: in the case of *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.

3.5 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.

TABLE 2
Android device's system Settings

Setting	Value
Adaptive Brightness	Manual - 78%
Bluetooth	Off
WiFi	On
Cellular	No SIM card
Location Services	Off
Auto-rotate screen	Off - Portrait
Zen mode	On - Total Silence
Pin/Pattern Lock Screen	Off
Don't Keep Activities	On
Account Sync	Off
Android Version	6.0.1

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 in-

11. More about "Don't Keep Activities" setting available at: https://goo.gl/SXkxVy (visited on September 10, 2019).

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:

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

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 Δt_i the duration of execution i.

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

$$Overhead(\%) = (\bar{x}'/\bar{x}'_{human} - 1) \times 100 \quad (2)$$

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 $\bar{x}'$ to compute the estimated energy consumption for a single interaction (*Sg*) as follows:

$$Sg = \bar{x}'/M \quad (3)$$

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[12], 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.

12. Project's *Github* repository: https://github.com/luiscruz/physalia-automators visited on September 10, 2019.

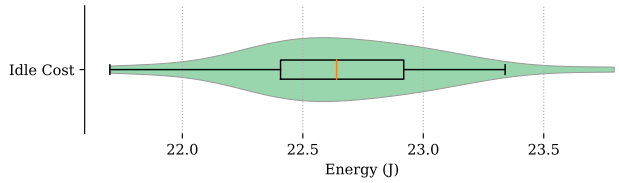

Fig. 3. Violin plot with distribution of the energy consumption of the app during 120 seconds.

TABLE 3
Descriptive statistics of *Tap* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	d	Overhead	#
Human	5.56	1.61	12.84	3.14	78.44		—	1
AndroidViewClient	19.71	0.21	42.10	11.75	293.86	8.65	274.6%	7
Appium	54.73	1.14	128.47	30.46	761.49	21.38	870.8%	9
Calabash	29.25	0.72	60.10	17.89	447.28	14.09	470.2%	8
Espresso	6.07	0.16	12.93	3.63	90.70	0.49	15.6%	2
Monkeyrunner	18.08	1.28	49.97	8.63	215.87	4.11	175.2%	5
PythonUiAutomator	9.15	0.54	18.93	5.57	139.32	2.24	77.6%	4
Robotium	14.59	4.00	25.63	9.74	243.57	2.11	210.5%	6
UiAutomator	7.64	0.55	17.77	4.28	107.03	1.13	36.5%	3

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 (Δt) in seconds, the mean energy consumption without idle cost ($\bar{x}'$, see Eq. 1), the estimated energy consumption for a single interaction (Sg, 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

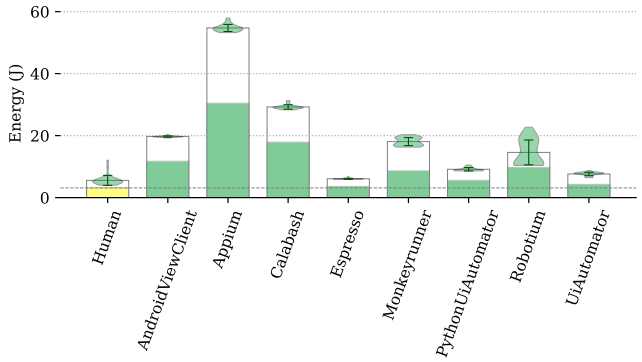

Fig. 4. Violin plot of the results for the energy consumption of *Tap*.

TABLE 4
Descriptive statistics of *Long Tap* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	d	Overhead	#
Human	13.33	2.21	32.86	7.12	177.92		—	1
AndroidViewClient	49.18	5.24	119.21	26.66	666.40	4.45	274.6%	7
Appium	25.34	0.86	49.60	15.96	399.08	7.75	124.3%	5
Calabash	46.96	1.81	94.27	29.14	728.57	12.44	309.5%	9
Espresso	19.87	0.54	37.00	12.88	321.94	6.20	80.9%	3
Monkeyrunner	21.68	0.74	48.07	12.60	315.04	5.57	77.1%	2
PythonUiAutomator	48.19	12.63	101.13	29.08	727.02	2.70	308.6%	8
Robotium	39.35	1.82	99.97	20.46	511.40	8.71	187.4%	6
UiAutomator	22.39	0.75	45.40	13.81	345.20	6.90	94.0%	4

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% ($\bar{x}' = 12.60$J) and 81% ($\bar{x}' = 12.88$J), respectively. *PythonUIAutomator* and *Calabash* are the most inefficient (overhead over 300%).

A remarkable observation is the efficiency of *Appium*'s *Long Tap* ($Sg = 0.40$J) when compared to its regular *Tap* ($Sg = 0.76$J). 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.

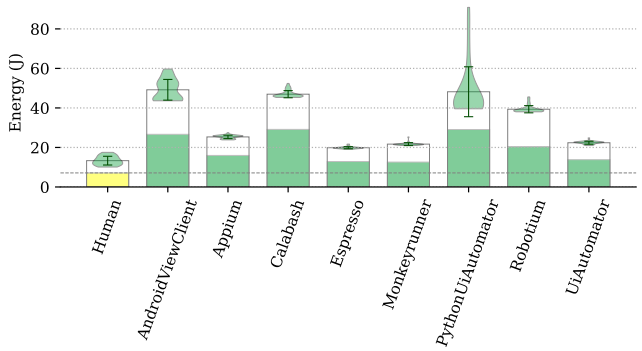

Fig. 5. Violin plot of the results for energy consumption of *Long Tap*.

TABLE 5
Descriptive statistics of *Drag and Drop* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	d	Overhead	#
Human	7.55	1.91	23.70	3.08	76.90		—	1
AndroidViewClient	21.31	0.76	62.15	9.57	239.24	6.67	211.1%	3
Appium	43.71	1.14	85.00	27.65	691.27	19.69	798.9%	7
Calabash	134.08	3.55	336.33	70.53	1763.27	26.79	2193.0%	8
Monkeyrunner	28.50	1.29	52.97	18.49	462.22	17.96	501.1%	5
PythonUiAutomator	36.30	3.77	93.53	18.62	465.56	5.54	505.4%	6
Robotium	20.63	1.02	52.17	10.77	269.29	7.75	250.2%	4
UiAutomator	14.48	0.63	30.27	8.76	219.02	8.19	184.8%	2

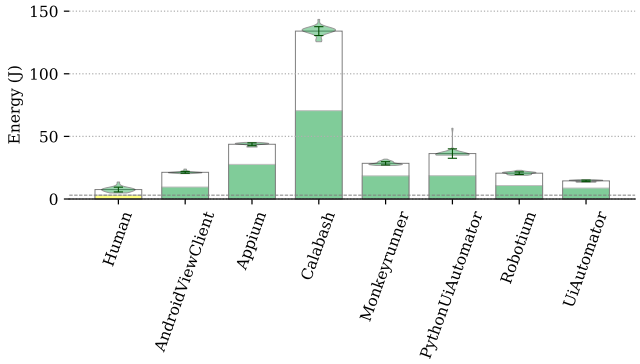

Fig. 6. Violin plot of the results for energy consumption of *Drag and Drop*.

4.4 *Drag and Drop*

Results for the interaction *Drag and Drop* are in Table 5 and Figure 6. *UIAutomator* is the best testing framework with an overhead of 185% ($\bar{x}' = 14.48$J). *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 6
Descriptive statistics of *Swipe* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	d	Overhead	#
Human	9.11	1.09	24.48	4.48	56.05		—	1
AndroidViewClient	45.29	0.62	115.87	23.39	292.41	27.93	421.7%	7
Appium	17.09	0.46	30.00	11.42	142.80	11.53	154.8%	3
Calabash	43.27	0.81	93.73	25.56	319.46	27.21	469.9%	9
Espresso	10.35	0.26	24.10	5.79	72.43	2.51	29.2%	2
Monkeyrunner	36.63	1.49	68.67	23.65	295.69	25.16	427.5%	8
PythonUiAutomator	26.42	0.91	54.60	16.11	201.32	14.05	259.1%	4
Robotium	41.30	0.67	96.00	23.16	289.46	26.85	416.4%	6
UiAutomator	27.56	0.65	60.13	16.20	202.49	19.93	261.2%	5

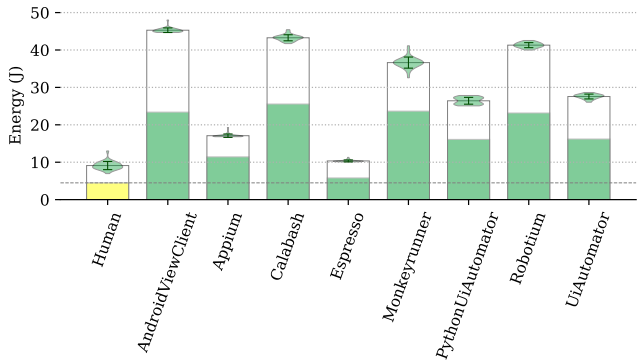

Fig. 7. Violin plot of the results for energy consumption of *Swipe*.

TABLE 7
Descriptive statistics of *Pinch and Spread* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	d	Overhead	#
Human	9.59	1.37	21.91	5.45	68.10		—	2
Calabash	41.31	8.66	81.93	25.83	322.83	4.82	374.0%	4
PythonUiAutomator	26.39	1.23	58.77	15.29	191.09	9.59	180.6%	3
UiAutomator	9.19	1.66	21.23	5.17	64.67	-0.21	-5.0%	1

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, it 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

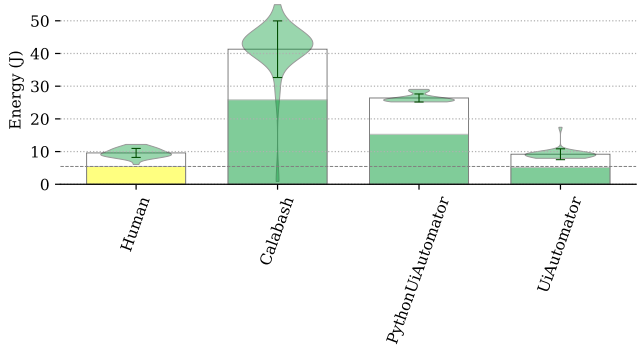

Fig. 8. Violin plot of the results for energy consumption of *Pinch and Spread*.

TABLE 8
Descriptive statistics of *Back Button* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	d	Overhead	#
Human	17.90	2.56	43.94	9.60	47.98		—	5
AndroidViewClient	85.75	2.11	179.73	51.79	258.94	17.79	439.7%	9
Appium	2.43	0.17	3.33	1.80	9.01	-8.33	-81.2%	2
Calabash	30.95	0.77	80.57	15.73	78.63	5.99	63.9%	8
Espresso	8.89	0.29	35.17	2.25	11.25	-7.66	-76.6%	4
Monkeyrunner	1.84	0.12	4.07	1.08	5.38	-9.13	-88.8%	1
PythonUiAutomator	53.62	1.78	220.03	12.04	60.20	1.68	25.5%	7
Robotium	60.44	5.18	308.10	2.22	11.10	-1.88	-76.9%	3
UiAutomator	49.87	1.03	208.20	10.53	52.64	0.83	9.7%	6

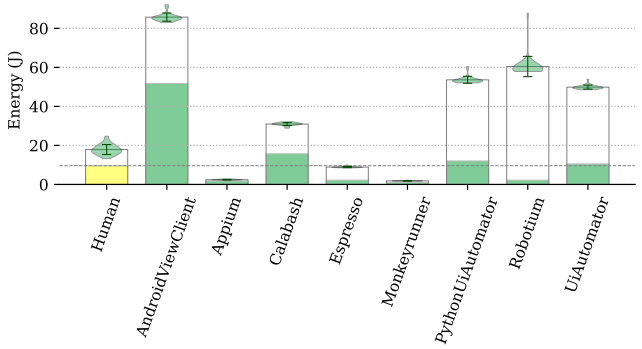

Fig. 9. Violin plot of the results for energy consumption of *Back Button*.

it all back to the initial state. Thus, the value for a single interaction (Sg) is the energy spent when this sequence of events is executed, but can hardly be extrapolated for other input interactions.

UIAutomator is the framework with the lowest energy consumption ($\bar{x}' = 1.42$J). 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 *UIAutomator*, *PythonUIAutomator*, and *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

TABLE 9
Descriptive statistics of *Input Text* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	d	Overhead	#
Human	22.11	4.06	54.09	11.89	1189.37		—	6
AndroidViewClient	222.08	4.31	523.37	123.18	12318.21	27.68	935.7%	9
Appium	44.43	1.89	105.27	24.54	2453.84	4.62	106.3%	8
Calabash	27.14	1.03	62.40	15.35	1534.70	1.40	29.0%	7
Espresso	6.83	0.18	14.03	4.18	417.96	-3.45	-64.9%	3
Monkeyrunner	6.18	0.29	8.03	4.67	466.58	-3.21	-60.8%	5
PythonUiAutomator	9.16	4.35	25.37	4.37	436.83	-2.07	-63.3%	4
Robotium	4.64	0.86	12.50	2.27	227.34	-4.26	-80.9%	2
UiAutomator	2.93	1.39	8.00	1.42	142.02	-4.62	-88.1%	1

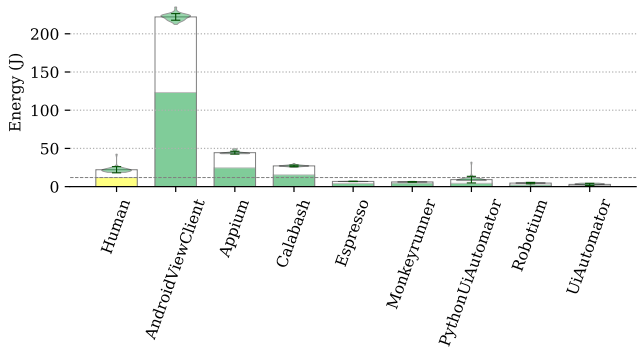

Fig. 10. Violin plot of the results for energy consumption of *Input Text*.

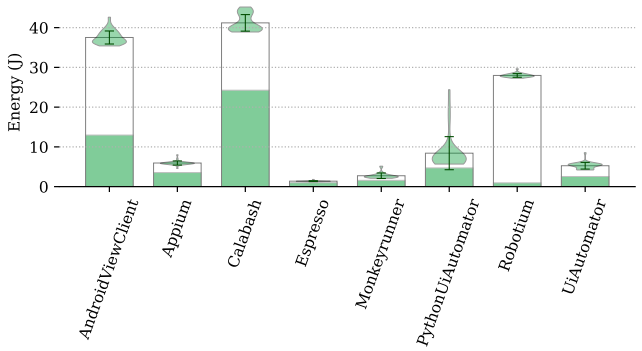

Fig. 11. Violin plot of the results for energy consumption of *Find by id*.

TABLE 10
Descriptive statistics of *Find by id* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	Rank
AndroidViewClient	37.52	1.64	129.91	12.97	46.34	7
Appium	5.94	0.51	12.73	3.53	12.62	5
Calabash	41.20	2.08	89.63	24.26	86.65	8
Espresso	1.37	0.11	2.03	0.99	3.54	2
Monkeyrunner	2.74	0.70	6.13	1.58	5.66	3
PythonUiAutomator	8.42	4.16	19.63	4.71	16.81	6
Robotium	27.97	0.46	143.03	0.94	3.37	1
UiAutomator	5.26	0.84	14.33	2.55	9.11	4

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 $0.94J$. 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 $27.97J$. 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.

TABLE 11
Descriptive statistics of *Find by description* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	Rank
AndroidViewClient	36.85	0.78	127.45	12.77	45.59	5
Appium	6.41	0.58	13.93	3.77	13.48	4
Calabash	41.41	7.02	88.20	24.75	88.38	6
Espresso	1.37	0.10	2.10	0.97	3.46	1
PythonUiAutomator	6.62	0.49	15.10	3.76	13.44	3
UiAutomator	5.13	0.61	14.47	2.40	8.57	2

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 in all components' text content. Since remote script-based frameworks, such as *Appium*, can do such task using the controller workstation, they can be more energy efficient from the DUT's perspective. For the same reason, *Find by content* has consistently higher energy usage than the other helper methods.

4.12 Statistical significance

As expected from previous work and corroborated with the violin plots, our measurements follow a normal distribution

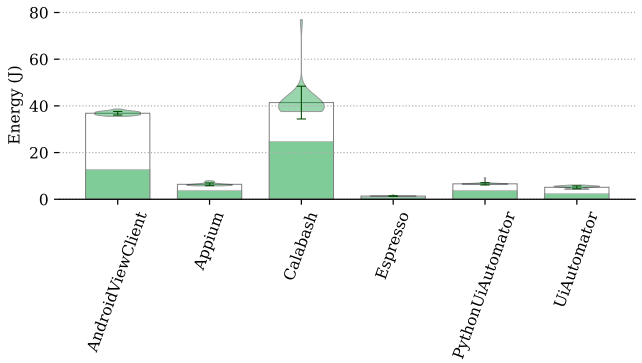

Fig. 12. Violin plot of the results for energy consumption of *Find by description*.

TABLE 12
Descriptive statistics of *Find by content* interaction.

	$\bar{x}$ (J)	s	Δt (s)	$\bar{x}'$ (J)	Sg (mJ)	Rank
AndroidViewClient	36.89	1.65	127.62	12.77	106.43	6
Appium	3.07	0.31	6.07	1.92	16.02	4
Calabash	31.77	4.64	79.63	16.72	139.35	7
Espresso	9.43	0.99	17.13	6.19	51.58	5
PythonUiAutomator	3.10	0.19	6.90	1.79	14.93	3
Robotium	23.74	0.48	124.90	0.14	1.15	1
UiAutomator	3.50	0.62	9.40	1.72	14.37	2

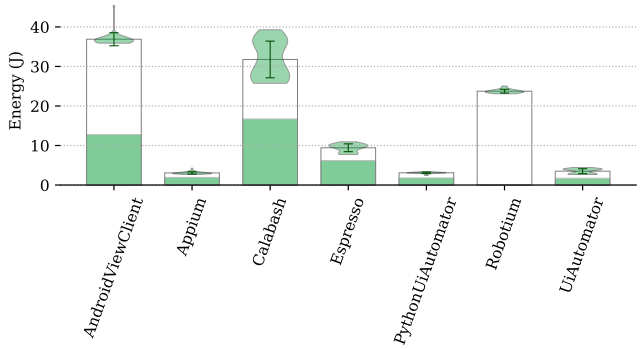

Fig. 13. Violin plot of the results for energy consumption of *Find by content*.

– also confirmed with the Shapiro-Wilk test. Thus, we assess the statistical significance of the mean difference of energy consumption between frameworks using the parametric 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[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*[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 58J, on average [8]. Considering our results, executing a single interaction such as *Drag and Drop* can increase energy consumption in 1.7J (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.

13. Project's *Github* repository: https://github.com/luiscruz/physalia-automators visited on September 10, 2019.

14. *Tail energy* is the energy spent during initialization or closure of a resource.

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) [11] 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

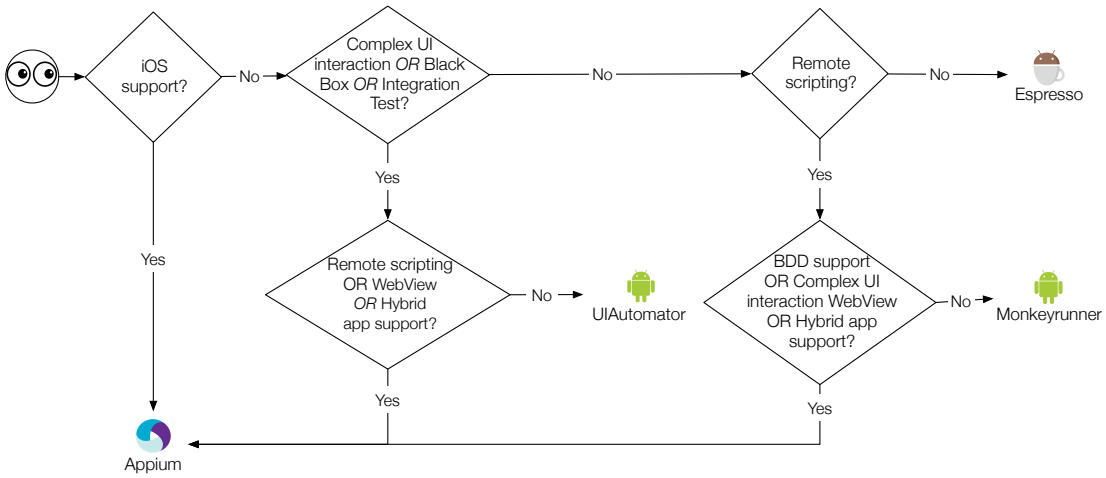

Fig. 14. Selecting the most suitable framework for energy measurements.

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 1.4J to 9.4J (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/EEA/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

[1] D. Ferreira, E. Ferreira, J. Goncalves, V. Kostakos, and A. K. Dey, "Revisiting human-battery interaction with an interactive battery interface," in *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. ACM, 2013, pp. 563–572.

[2] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[3] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "Helping programmers improve the energy efficiency of source code," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 238–240.

[4] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17), page to appear*, 2017.

[5] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, 3 2019.

[6] I. C. Morgado and A. C. Paiva, "The impact tool: testing UI patterns on mobile applications," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 876–881.

[7] R. M. Moreira, A. C. Paiva, and A. Memon, "A pattern-based approach for GUI modeling and testing," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 288–297.

[8] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of android applications," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 121–130.

[9] S. Lee, W. Jung, Y. Chon, and H. Cha, "Entrack: a system facility for analyzing energy consumption of android system services," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 191–202.

[10] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 2–11.

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

[11] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?" in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 103–114.
[12] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 115–126.
[13] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi, "Deconstructing the energy consumption of the mobile page load," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 1, no. 1, pp. 6:1–6:25, 2017.
[14] K. Rasmussen, A. Wilson, and A. Hindle, "Green mining: energy consumption of advertisement blocking methods," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM, 2014, pp. 38–45.
[15] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirović, "Assessing the impact of service workers on the energy efficiency of progressive web apps," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2017, pp. 35–45.
[16] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *IEEE/ACM International Conference on Mobile Software Engineering and Systems, MobileSoft 2017*, 2017, pp. 46–57.
[17] ——, "Using automatic refactoring to improve energy efficiency of android apps," in *XXI Ibero-American Conference on Software Engineering (CIbSE, Best Paper Award)*, 2018.
[18] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, 2016, pp. 59–69.
[19] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 72–81.
[20] C. Sahin, L. Pollock, and J. Clause, "From benchmarks to real apps: Exploring the energy impacts of performance-directed changes," *Journal of Systems and Software*, vol. 117, pp. 307–316, 2016.
[21] R. Coppola, E. Raffero, and M. Torchiano, "Automated mobile UI test fragility: an exploratory assessment study on android," in *Proceedings of the 2nd International Workshop on User Interface Test Automation*. ACM, 2016, pp. 11–20.
[22] R. Coppola, "Fragility and evolution of android test suites," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 405–408.
[23] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "Sitar: GUI test script repair," *Ieee transactions on software engineering*, vol. 42, no. 2, pp. 170–186, 2016.
[24] S. Gunasekaran and V. Bargavi, "Survey on automation testing tools for mobile applications," *International Journal of Advanced Engineering Research and Science*, vol. 2, no. 11, pp. 2349–6495, 2015.
[25] M. K. Kulkarni and A. Soumya, "Deployment of calabash automation framework to analyze the performance of an android application," *Journal 4 Research*, vol. 2, no. 03, pp. 70–75, 2016.
[26] K.-C. Liu, Y.-Y. Lai, and C.-H. Wu, "A mechanism of reliable and standalone script generator on android," in *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 372–374.
[27] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17), page to appear*, 2017.
[28] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 429–440.
[29] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.
[30] M. Linares-Vásquez, "Enabling testing of android apps," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2. IEEE, 2015, pp. 763–765.
[31] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 204–217.
[32] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.
[33] C.-H. Liu, C.-Y. Lu, S.-J. Cheng, K.-Y. Chang, Y.-C. Hsiao, and W.-M. Chu, "Capture-replay testing for android applications," in *Computer, Consumer and Control (IS3C), 2014 International Symposium on*. IEEE, 2014, pp. 1129–1132.
[34] D. Li and W. G. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM, 2014, pp. 46–53.
[35] D. Li, A. H. Tran, and W. G. Halfond, "Making web applications more energy efficient for oled smartphones," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 527–538.
[36] A. Hindle, "Green mining: a methodology of relating software change and configuration to power consumption," *Empirical Software Engineering*, vol. 20, no. 2, pp. 374–409, 2015.
[37] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, 2016, pp. 139–150.
[38] A. Banerjee, H.-F. Guo, and A. Roychoudhury, "Debugging energy-efficiency related field failures in mobile apps," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, 2016, pp. 127–138.
[39] R. Saborido, V. V. Arnaoudova, G. Beltrame, F. Khomh, and G. Antoniol, "On the impact of sampling frequency on software energy measurements," PeerJ PrePrints, Tech. Rep., 2015.
[40] S. Abdulsalam, Z. Zong, Q. Gu, and M. Qiu, "Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency," in *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, Dec 2015, pp. 1–8.
[41] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 15–24.

Luís Cruz holds a Ph.D. in Computer Science from the University of Porto, Portugal. He is currently a postdoc researcher in Delft University of Technology where most of his research is carried out. His main research fields are Mobile Software Engineering, Green Computing, and Mining Software Repositories. His work aims at improving mobile development processes concerning the energy consumption of mobile applications.

Rui Abreu holds a Ph.D. in Computer Science - Software Engineering from the Delft University of Technology, The Netherlands, and a M.Sc. in Computer and Systems Engineering from the University of Minho, Portugal. His research revolves around software quality, with emphasis in automating the testing and debugging phases of the software development life-cycle as well as self-adaptation. Dr. Abreu has extensive expertise in both static and dynamic analysis algorithms for improving software quality. He is the recipient of 6 Best Paper Awards, and his work has attracted considerable attention. Before joined IST, ULisbon as an Associate Professor and INESC-ID as a Senior Researcher, he was a member of the Model-Based Reasoning group at PARC's System and Sciences Laboratory and an Assistant Professor at the University of Porto. He has co-founded DashDash in January 2017, a platform to create web apps using only spreadsheet skills. The company has secured $9M in Series A funding in May 2018.