Greenspecting Android Virtual Keyboards

Rui Rua
HASLab/INESC TEC, Portugal
University of Minho, Portugal
rui.a.rua@inesctec.pt

Tiago Fraga
HASLab/INESC TEC, Portugal
University of Minho, Portugal
tiago.m.santos@inesctec.pt

Marco Couto
HASLab/INESC TEC, Portugal
University of Minho, Portugal
marco.l.couto@inesctec.pt

João Saraiva
HASLab/INESC TEC, Portugal
University of Minho, Portugal
saraiva@di.uminho.pt

ABSTRACT

During this still increasing mobile devices proliferation age, much of human-computer interaction involves text input, and the task of typing text is provided via virtual keyboards. In a mobile setting, energy consumption is a key concern for both hardware manufacturers and software developers. Virtual keyboards are software applications, and thus, inefficient applications have negative impact on the overall energy consumption of the underlying device.

Energy consumption analysis and optimization of mobile software is a recent and active area of research. Surprisingly, there is no study analysing the energy efficiency of the most used software keyboards and evaluating the performance advantage of its features. In this paper we studied the energy performance of five of the most used virtual keyboards in the Android ecosystem. We measure and analyse the energy consumption in different keyboard scenarios, namely with or without using word prediction. This work presents the results of two studies: one where we instructed the keyboards to simulate the writing of a predefined input text, and another where we performed an empirical study with real users writing the same text.

Our studies show that there exists relevant performance differences among the most used keyboards of the considered ecosystem, and it is possible to save nearly 18% of energy by replacing the most used keyboard in Android by the most efficient one. We also showed that is possible to save both energy and time by disabling keyboard intrinsic features and that the use of word suggestions not always compensate for energy and time.

KEYWORDS

Android, Green Software, Keyboard, Energy

[1] This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. The first author is also financed by FCT grant SFRH/BD/132485/2017.

MobileSoft 2020, May 2020, Seoul, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Rui Rua, Tiago Fraga, Marco Couto, and João Saraiva. 2020. Greenspecting Android Virtual Keyboards. In *Proceedings of ACM Conference (MobileSoft 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the previous century, software runtime was one of the major concerns for language designers, compiler writers, and software developers. Nowadays, energy consumption has become one of the main concerns both for hardware manufacturers and software developers. In fact, the widely use, and still increasing number of very powerful mobile computing devices requires energy efficient hardware and software. Green software community has been developing techniques and tools to address the energy oriented decision making, which is being demanded from developers [26]. For example, researchers studied which programming practises affect energy consumption [9, 12, 13, 18, 28], the energy overhead of Android testing [11] or development [8] frameworks.

Very much like a computer user relies on a physical keyboard to interact with a wired computer, a mobile device user mostly relies on a virtual keyboard to interact with a mobile device. A virtual keyboard is a software application offering advanced features that overcome regular physical keyboards, such as, multilingual word prediction and auto-correction, key animations, among others. Although mobile operating systems offer a predefined keyboard, there is a large number alternatives offering similar features which users can install on their devices. Being a software application, virtual keyboards may influence the overall energy consumption of the underlying mobile device. Surprisingly, there is no research work on performing a detailed analysis of the energy efficiency of the mainly used keyboards, nor on analysing the energy impact of their advanced features.

In this paper we analyse the energy consumption of widely used virtual keyboards and their advanced features. We consider the five most used keyboards available in the Android ecosystem, which is the widest used mobile phone operating system worldwide. To analyse its energy consumption we conducted two studies: First, we used an Android testing framework to exercise the virtual keyboards, by automatically writing a predefined input text in the same mobile device, with exactly the same conditions, while monitoring the energy consumption. Next we conducted a second study with 40 real users who also had to write the same/previous input text. Moreover, in both studies we considered two usage scenarios.

In the former, we considered one with all features off and another in default mode (the pre-install setup, where prediction and auto correction are available). For the latter, we considered similar scenarios but in default mode: One without using the word suggestions and another maximizing the use of such feature.

These studies aim at answering the following research questions:

- **RQ1:** Are there any significant differences in energy consumption between the most used Android keyboards?
- **RQ2:** Can significant energy gains be achieved by minimizing the work and features that keyboards use?
- **RQ3:** Does the eventual runtime gain of using (computational intensive) word prediction algorithms compensate for energy?

The results of our studies show that there is a statistically supported difference in the energy consumed by the most used keyboards in the Android platform. We also show that the energy gains can go up to 18% just by replacing the most energetically inefficient keyboard with the most efficient one. We also show that there are cases where minimizing the number of features that keyboards offer can help achieve a drop in energy consumption of up to 9.3%. The preliminary results of the second study show that the use of suggestions may not always be beneficial in saving time and energy.

The remaining article follows the following structure: section 2 introduces the keyboards select for this study and describes the procedure followed to automatically test and monitor the keyboards in real physical devices. It also presents the methodology followed to isolate the testing environment and the system image from undesired interference. Section 3 presents the results obtained from the studies performed automatically and with real users and section 4 presents the main conclusions that the results allow to extract. In order to discuss some of the potential threats to this work, in 5 we discuss some of the decisions followed to elaborate this work. Finally, in section 6 we present some of the work related to this study and in 7 we draw the main conclusions that can be inferred from this work and the respective results.

2 STUDYING THE ENERGY EFFICIENCY OF VIRTUAL KEYBOARDS

In this section, we fully explain all the underlying aspects of the conducted studies to analyze the energy consumption of Android keyboards. First, we explain how the keyboards used for the experiment were selected, and we describe each one of them (Section 2.1). In Section 2.2, we then explain the full experimental setup: the devices used for the tests, their state, the supplementary tools used and their purpose, also addressing some of the precautions we took to minimize external interference in the procedures followed in both studies. We present in Section 2.3 the full details of the procedure followed for the experiment which includes 2 different studies: one performed using an automatic approach and another using real users.

2.1 Keyboards Under Test

A virtual keyboard application is an essential software artifact for mobile device users. A keyboard is not a standalone application, but instead is the software used as the input method for other software applications, for example, to write text messages, edit texts, write an e-mail, fill forms, etc. In fact, a virtual keyboard application offers advanced features that greatly overcome classical physical keyboards, such as the use of multilingual auto-correction and word prediction, key pressing animations (both visual and vibration ones), etc. Thus, the keyboard app became a swiss-army knife that greatly contributes to improve the user experience of a mobile device/ecosystem.

To provide those advanced features, virtual keyboards are complex software applications. For the language specific auto-correction feature the software needs to efficiently index large dictionaries. Moreover, to give helpful word predictions, keyboards rely on learning algorithms and artificial intelligence techniques. We can look at nowadays keyboards as large and complex software systems. For example, *GBoard* keyboard (Google keyboard) application is written in several programming languages and in its version 4.1.2.3 it has about 265,104 lines of code as shown in Table 1.

Language	#Files	#Lines	LoC
Java	664	117095	75181
C/C++	127	1923	14543
Bash	3	137	121
HTML/CSS	7	392	77
C/C++ Header	158	15788	9941
XML	1338	241308	164727
Make	1	8	9
Gradle	1	110	87
Others	8	491	145
Total:	**2307**	**377252**	**265104**

Table 1: GBoard: Languages, number of files and (source code) lines

Being virtual keyboards an important aspect of any mobile device ecosystem, a predefined one is pre-installed by all device manufacturers. The keyboard that typically comes with the Android platform is the GBoard keyboard. However some manufacturers tend to offer alternative keyboards (for example, Samsung pre-installs its own keyboard, while Huawei re-uses Microsoft *Swiftkey* keyboard [1]). Not only these large companies provide such advanced virtual keyboard, there is a huge number of similar keyboard applications available from the official Android Play Store.

Given the wide variety of keyboard applications, and the unfeasibility of analyzing the energy consumption of each on individually, we selected a representative subset of the keyboard apps available on the Play Store. First, we used the provided number of downloads to select the 10 most downloaded keyboards. Since the number of downloads does not necessary mean the keyboard is currently being used, we combined this information with the data contained in the GreenHub [20] dataset: an ongoing green software crowd-sourcing initiative, that contains data collected from

[1] https://www.microsoft.com/en-us/swiftkey

more than 60,000 currently running devices, which included the app packages installed in each of them. Using the GrenHub dataset we select the five most used keyboards from the previous set of 10. Samsung keyboard was excluded from this set because it is only available for Samsung devices, which do not support our energy monitoring framework (see Section 2.2), and its applicability is limited compared to the other alternatives. Table 2 shows the five keyboards we consider in our empirical studies, specifying the Android package, the app version, number of downloads, and whether are available in other mobile platforms.

	Cheetah	Fancykey	Go	Google	Swiftkey
Auto-Capitalization	E	E	E	E	E
Suggestions display	E	E	E*	E	E
Next word prediction	E	E	E	E	E
Auto-correction	E	E	E	E	E
Emoji prediction	✗	E	E	✗	D
Gestures (swipe input)	E	E	E	E	E
Multi-dictionary	E	E	E	E	E**
Search engine integration	E	E	E	✗	E
Keypress animations	E	E	E	E	E
Voice input	E	E	E	E	E
Keypress Vibration	E	E	D	E	E
Keypress Sound	E	E	E	D	D

Table 2: KUT features list. E - Enabled by default, D - Disables by default.

2.2 Experimental Setup

Having defined the five representative virtual keyboards we will consider in our study, we now present the full setup designed for our experiments: the devices used for the tests, the tools used to *(i)* precisely monitor the energy consumed by the keyboards and *(ii)* to define the same tasks being executed over different keyboards, and how we guarantee that we minimize, as much as possible, all possible external interference to the keyboard application usage.

Mobile Device and Tailored Android: This study was conducted on two rooted, fully-charged, LG Nexus 5 devices. Google supported upgrades from the platform version on this model up to version 6.0.1, and it was this version that was chosen to perform the study. The version 6 of the platform is still the most widely used version worldwide [1], and it uses the ART (Android RunTime) virtual machine, that compiles applications at install time, using AOT (Ahead Of Time) compilation.

To have accurate energy measurements that reflected only the computational effort of keyboards, we reduced Android to the minimal number of processes possible, so that there are no non-necessary processes consuming energy during the studies. In fact, the considered Nexus 5 devices both run a customized boot image, obtained by performing root-access changes in the 6.0.1 official version, which was reduced to the minimal set of apps/processes needed to conduct our study, being able to reduce the minimum number of processes running from 8 to 3. Moreover, we restored the boot image between different keyboard executions.

At the system level, precautions were taken to ensure that all external communication interfaces were disconnected. Thus, no SIM cards were inserted in the devices, and all network and sensor interfaces (Wi-Fi, Bluetooth, GPS, NFC, etc.) were turned off, as well as the speakers. Furthermore, we also executed the AUT (App Under Test) in immersive mode, in order to hide the system bar and minimize the impact of eventual events animations associated with it. All tests were also performed with the screen at the same brightness level.

Energy Measurement: Measuring software power consumption is a nontrivial process [25], also because of the difficulty in reducing external interference (commonly called the Hawthorne effect) to the process to be monitored. Since this work aims to estimate the consumption in order to minimize this interference, some precautions had to be taken, from the system image where the tests were performed to the Profiler configuration and the testing procedure itself. There are many factors that might have impact the performance of the KUT and the system itself, that can be internal or external to the keyboard application. Table 3 reports some of these factors that we identified and tried to minimize.

External	Internal	
Hardware	Auto-correction	Suggestions
OS version	Sound and Vibration	Emojis
Typing rate	Theme/Layout	Multi-Language support
System Load	Animations	Auto-capitalization
System Language	Gestures	Other user preferences

Table 3: External and internal factors that can impact keyboard performance

To measure the energy consumption of the virtual keyboards we used the Trepn energy profiler [2]. Although there are currently several alternatives to estimate and measure the energy consumption of applications on the Android [6, 9, 22] platform, we have selected Trepn since it adapts perfectly to the device and the procedure we intend to implement. This tool is a software-based solution developed by Qualcomm, that works on devices with Snapdragon chipset-based running Android, as is the case for our two Nexus 5. Trepn can be used to profile hardware usage (like GPS, WiFi and others), resources usage (memory, CPU) and power consumption of both the system or standalone Android applications. This system does not need external hardware devices, as it gets its power readings from the power management Integrated Circuit (PMIC) and the battery fuel gauge software. Moreover, Trepn reportedly produces accurate estimation [3], virtually identical to those obtained by hardware-based solutions, such as Monsoon [3].

Input Text and Test Application: Because keyboards are not standalone applications, we needed a widely used application where the keyboard is the main input textual method. To reduce any possible noise due to background operations, this application should, however, have a limited number of features, and not rely on (for instance) network interactions or animations. Therefore, as AUT we

[2]Trepn Profiler: https://developer.qualcomm.com/software/trepn-power-profiler
[3]Monsoon: https://www.msoon.com

chose the *Wordpad* application [4], which has a simple and clean interface, and can be used offline without any background task being constantly doing uncontrolled/unpredictable work, like searching for an internet connection.

As input text, we randomly sampled 100 words from a paragraph from the book "Harry Potter and the Goblet of Fire" [27]. This text consists in words with an average length of 6 characters and contains no punctuation. We removed the punctuation from the text since many keyboards have different behaviours when a dot or comma are pressed. Also, in a preliminary evaluation of the real users study, they complained that the fact of the selected keyboards had different places and layouts to show and select the required punctuation and that affected their writing performance.

Automate Keyboard Inputs: In order to exercise the five keyboards, we needed to define a strategy for automatically writing a pre-defined text input using each keyboard. Therefore, we stored the input text in an auxiliary file, and searched for a tool that could read its content and simulate writing it on *Wordpad*. For this purpose, we used the Android View Client (AVC) framework[5]: a Python-based tool that evolved from *monkeyrunner* [2], and which allows to manage and interact with multiple devices connected to the workstation. This tool has been used in similar works before [10], and uses ADB and the *input* command [6] to simulating touches, taps, swipes and text input, among other events events.

Running Keyboard and Collecting Energy Metrics: Finally, for developing the testing workflow, we reused the AnaDroid framework [29]. We reused its tools and workflow to define the execution procedure, gather relevant metrics during the execution of the tests and generate reports from the profiled data obtained by Trepn.

2.3 Experimental Procedure

The energy consumption of each of the considered keyboards was studied using two approaches: *(i)* using a fully automated approach based on the tools described before, and *(ii)* by performing an empirical study, with real users that tested the keyboards under controlled conditions. We chose these two approaches since we aimed at determining the potential gains that could arise from choosing the most energy efficient keyboard, but also to study the impact that the human factor has on such gain.

2.3.1 Automated Approach

As stated in section 2.1, virtual keyboards offer a wide range of features, but the core component is typically the same, as is the possibility to turn on/off more advanced features. From animations, sounds, emojis, and gestures, to complex natural language processing mechanisms (with word suggestions and correction features), all the considered keyboards offer these features with more or less complexity.

Given the high level of customization that this type of application offers, it was necessary to determine real usage scenarios that could be replicated on all keyboards, and in which it was fair to establish comparisons between them. Thus, two different usage modes have been defined:

- **Minimal Mode:** In this mode all possible features are turned off. This usage scenario intends to represent the most minimalist execution scenario possible, which resembles the functionality offered by a physical keyboard, and that reduces to the minimum the computational effort of the keyboard application. It also assures that all keyboards are tested under the same conditions.
- **Default Mode:** This mode represents a usage scenario where the enabled features are the ones provided by each keyboard at the moment of its installation. It contains the features that developers defined for that version, which is the ideal keyboard execution scenario. The features activated by default, for all of our five keyboards, include word prediction and suggestion, auto-correction, among others.

In order to configure keyboards in the minimal mode, it is necessary to manually disable each of the features, using the KUT settings. Table 2 shows the features that are automatically activated in default mode for each of our five keyboards: they are marked with $'E'$. The single exception is the Swiftkey keyboard, which needs an internet connection to download the dictionary of the selected language and manually activate it. For this particular case, we use an Wi-fi connection to download dictionary and then we rebooted the imaged and wiped the device caches. Since the customized image doesn't contain another services (like Google Play Services or a browser) in the background waiting for network connections, we insured that no other resources were downloaded.

To perform our first study, where the five keyboards are exercised both in minimal and default mode within the Wordpad app, we use the methodology specified in Algorithm 1.

Algorithm 1 Text input simulation algorithm

```
loadBootImage()
for k : keyboards do
    installKeyboard(k)
    if mode == "minimal" then
        manuallyDisableFeatures()
    end if
    for i=0;i<25;i++ do
        text = loadTextToWrite(textfile)
        wordsCoords = loadKeyboardKeyCoordinates(text)
        wipeWordpadCache()
        initprofiler()
        openWordpad()
        openKeyboard(k)
        logDeviceState()
        startProfiler()
        writeTextInWordpad(k, wordsCoords )
        stopProfiler()
        logDeviceState()
        closeKeyboard()
        closeWordpad()
        exportProfiledData()
        shutdownProfiler()
    end for
    restoreBootImage()
    cleanDalvikArtCache()
end for
generateResults()
```

[4] Wordpad application: shorturl.at/qsuC5
[5] AVC: https://github.com/dtmilano/AndroidViewClient
[6] Input tool: shorturl.at/jlsCN

As discussed in Section 2.2, we use a simpler Android boot image, which reduced to the minimum the number of background processes. Thus, we start by loading that boot image to the Nexus device. Then, for each of the (five) keyboards, a keyboard is installed in the considered mode (minimal or default). If in minimal mode, it is requested an additional manual configuration to turn off advanced features. After having the KUT installed and configured, the input text is loaded, along with the mapping between each letter and the corresponding coordinates in the KUT. The use of this pre-calculated key coordinates for each keyboard is required, although the Input program offers the possibility to write text without explicitly sending the coordinates of each key. However, this option doesn't correspond to the option of pressing individually each key with a touch, leading to different keyboard behaviours (for instance, not showing keypress animation or not performing auto-spacing when touching the suggestion bar).

One everything is settled, we repeat the text writing 25 times using the KUT within *Wordpad*. Each of these executions consists of the following steps: the *Wordpad* cache is cleaned, to ensure that there is no text related to previous tests that may cause biased data. The Trepn service is then started, and the Wordpad is initialized. The keyboard app opens and the device resource status is collected (number of processes running, CPU and RAM usage, among others) just before starting the monitoring process ('*startProfiler ()*'). This step consists of using the Trepn states to temporarily limit the beginning of the process. The next step consists of automatically writing the text. For this step, the *Input* program is used through ADB, so that each key is pressed to form the words of the input text. Finally, the monitoring process is finished and the device status is recorded again. The AUT and KUT are closed, the data from the monitoring session is exported to a *csv* file, the profiler is turned off, and the data is exported to the workstation. Finally, after the 25 executions of a keyboard, the boot image is restored and the device's caches are cleaned.

2.3.2 *Empirical User Study*

As described in the previous section, it is possible to instruct a virtual keyboard to write a pre-defined input text. When exploring the energy impact of keyboards advanced features, however, a similar automated process would be infeasible to apply in all five keyboards. For example, automating the word prediction feature is a complex task: keyboards use different strategies for the displaying word suggestions: some only show three predictions, while others display a pull-down list, sometimes in different positions withing the keyboard. The position of the best suggestion within the list also varies between keyboards, thus increasing even the complexity and computation overhead of an automated approach.

Thus, to rigorously study the energy impact of these features, we performed an empirical study with 40 real users. The volunteers that participated in our study were aged between 21 and 36, with 17.5% being females and 82.5% males. Only 1 volunteer has the same mobile device and 20% of them used one of the keyboards evaluated in our study, the Swiftkey keyboard. 75% of users run their device predefined keyboard, and 87.5 use the Android operating system. The wide variety of keyboards used by volunteers is valuable for our study. This ensures that they are unconsciously not privileging a specific keyboard, either because they are used

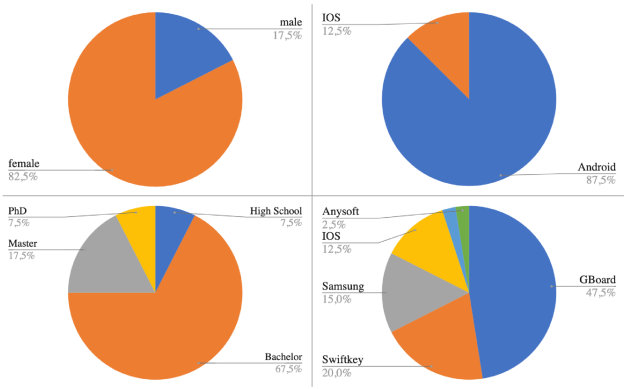

Figure 1: Statistics of the volunteers used in our study

to that keyboard or because of personal preference. The main statistics regarding the volunteers profile of this study is presented in figure 1.

For our study with real users, we considered the two best keyboards in terms of energy performance from the previously described study. Moreover, we divided the users into 4 groups of ten volunteers, so that each group evaluates 1 keyboard in a specific execution scenario. The two scenarios are:

- **Minimizing Suggestions:** For each of the keyboards, 2 distinct groups of users are asked to transcribe a given text, without resorting to the use of suggestions. With keyboards in the default state, users should write all words. This mode aims to represent the scenario in which the keyboard user prefers to write the whole word instead of using the suggestions.
- **Maximizing suggestions:** For both keyboards, groups of 10 users have to transcribe a given text, while maximizing the use of keyboard suggestions, which is in default mode. This mode aims to represent the scenario in which the user prefers to always look for the suggestion instead of having to write the full word.

The process was replicated in the same Nexus device used in Section 2, and every user transcribed the exact same text. In order to have a more strict control over what users were doing before and during the process (like starting to type before the start of the profiler) the same setup as in the procedure referred to in 1 was used, with the necessary changes. The resultant procedure is described with algorithm 2. The main differences are the fact that 2 boot images were used, almost the same used in the automatic study, but each one containing one keyboard pre-installed and ready to use. The image and caches were restored between each test, to prevent the keyboard's prediction algorithm to adapt and possibly overfit the input text. The start and end of the profiler is now delimited directly by the user and the step '*writeTextInWordpad()*' has been replaced by the process of the user writing the required words.

Finally, the naive automatic procedure developed to simulate typing text and using suggestions is also very similar to the one developed described in algorithm 1. Since most of these keyboards (with the exception of GBoard) are not open-source, without instrumenting the source code or having access to the view-hierarchy, it

Algorithm 2 Real User Text input algorithm

```
for k : {GO, Swiftkey} do
    loadBootImage(k)
    for u : users do
        wipeWordpadCache()
        initprofiler()
        openWordpad()
        openKeyboard(k)
        logDeviceState()
        startProfiler()
        realUserTextInput()
        stopProfiler()
        logDeviceState()
        closeKeyboard()
        closeWordpad()
        shutdownProfiler()
        exportProfiledData()
        restoreBootImage(k)
        cleanDalvikArtCache()
    end for
end for
generateResults()
```

is not possible to obtain the suggested words in realtime and with negligible computational cost. Thus, we decided to manually measure the effectiveness of each keyboard prediction mechanism in one single test and store this information in a map data structure. For each word of a given text, we associated the target word to the number of characters it was necessary to type in order to appear the word in the suggestions bar. We also evaluated for each keyboard the approximate coordinates of the place where typically the most probable suggestion appears. Using this pre-calculated information, we performed 25 tests using the same procedure of algorithm 1, but with a different text input procedure, that can be resumed in the algorithm 3.

Algorithm 3 Naive text input algorithm

```
textInput (wordMap, suggestionCoords) {
    for word, nCharsToWrite : wordMap do
        for i=0;i<nCharsToWrite; i++ do
            typeChar( word[i])
        end for
        touchSuggestionBar(suggestionCoords)
    end for
}
```

3 ANALYSING THE ENERGY CONSUMPTION

In this section we present in detail the results obtained in the studies described in the previous sections. Section 3.1 contains the results obtained with the automated study, while section 3.2 presents the results obtained with the empirical study with real users.

3.1 Default and Minimal mode

The overall results for the automated experiment are depicted in Figures 2 and 3. The former illustrates the energy consumption for all 5 keyboards, while the latter contains the values for the elapsed time. Each pair of box plots serves as a comparison between the 25 test executions of both testing modes: default (with only the default features of each keyboard enabled) and minimal (with all features disabled).

These plots show statistical differences among KUT energy consumption. The keyboards with lower median and average values of energy consumed in default mode were Go (median of $724.6J$ and average of $687.5J$) and Swiftkey (median of $727.9J$ and average of $715.98J$). Gboard and Cheetah are the most energy-greedy keyboards, with median values of $890.7J$ and $816.1J$.

When looking at the results for the minimal mode, the most energy efficient keyboard were Cheetah and Swiftkey, with median values of $740.2J$ and $756.8J$, and average values of $757.2J$ and $757.6J$, respectively. Swiftkey appears as one of the most energy efficient keyboards, independently of the testing mode. On the other hand, the Cheetah keyboard was by far the one that benefited more with feature disabling, while the Go keyboard actually increased energy consumption with it, being the keyboard with highest energy consumption in the minimal mode.

Examining the elapsed time values, illustrated in Figure 3, together with the energy consumption values, allow us to verify how much related is the energy efficiency with performance. We can confirm that, for the minimal mode, every KUT obtained lower median value that the ones obtained in the default mode. The most performance efficient keyboards were GBoard in the default mode and GO in minimal mode, which were also the most energy-greedy ones in the same modes. The opposite occurs, for instance, for Cheetah in minimalist mode or Swiftkey in default mode, being that in these modes they were the most energy efficient, but in terms of performance they were the most inefficient.

As mentioned in Section 2.3, a wide range of metrics were collected during the execution of the tests. In order to assess the potential impact that each of them may have on energy consumption and even on the consumption of other results, the Figure 4 presents a correlation matrix that aims to illustrate these relationships. The information from the tests performed in a single dataset was aggregated and the metrics that had no relation between the others were filtered. In this figure it can be observed that the features more directly correlated (marked with a deeper blue tone) with the energy consumed are CPU 1 frequency, CPU 2 frequency, CPU Load Normalized (load normalized across all cores, according to the maximum CPU frequency) and battery power. The least correlated features are CPU 2 and 4 load, CPU load (across all cores) and elapsed time (coinciding with the data observed in Figures 2 and 3).

3.2 Real User Interaction and Simulation

Following the same structure of the previous section 3.1, for this study we also present the main results with box plots. However, we only show the results for the 2 keyboards that are cross-platform and were the most energy efficient in the default mode, according to the data contained in the Figure 2: Swiftkey and GO keyboard. Since we wanted to test each mode several times, if we considered all of the 5 keyboards, we would end up with only 4 tests for each mode. Furthermore, we decided to select the two most efficient in default mode. The results of the energy consumed and execution

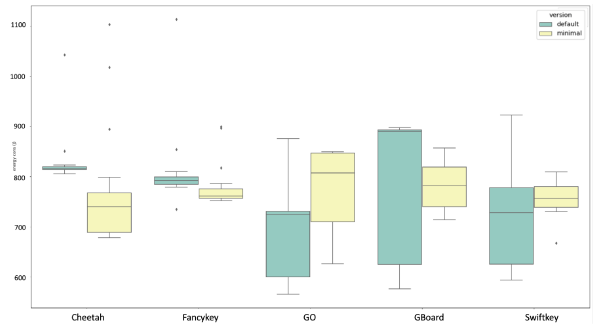

Figure 2: Energy consumption box plots of all keyboards

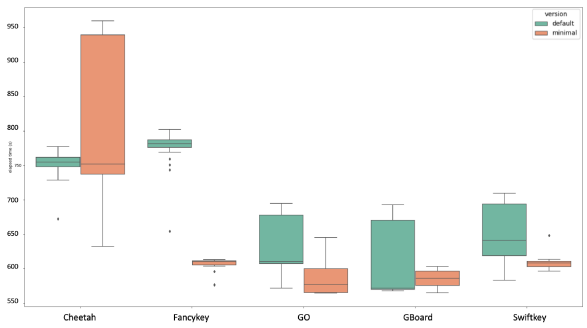

Figure 3: Tests Elapsed time of all keyboards

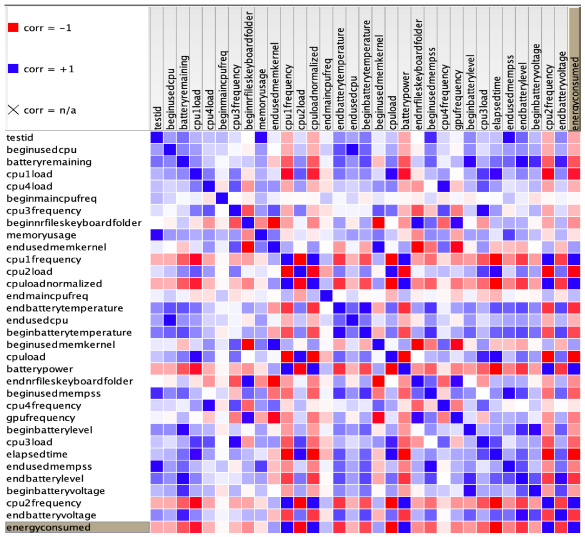

Figure 4: Correlation matrix of the sampled data during tests execution

time in each of the 10 executions of each configuration can be seen in Figures 5 and 6.

Regarding energy consumption, we can see that using more features (in this case only word prediction) produces different results on these 2 chosen keyboards, similarly to what happened in the automated study (depicted in Figure 2). According to the results, looking at the median, the energy consumption slightly increases when using suggestions in the GO keyboard, but not according to the 1st and 3rd quartile. It also led to users taking more time to complete their task, as the values in Figure 6 suggest. As for the Swiftkey keyboard, the suggestions mode was the most energy efficient looking at the median value, but also the most time consuming.

The automatic naive procedure was also performed, in order to simulate the use of word suggestions by real users. This experiment, which was performed with the default features enabled, had quite different results from those obtained with real users, as can be seen in Figures 7 and 8. Using this automatic procedure, looking at the median values obtained for each of the keyboards, there was an average gain of $311.74J$ (a gain of 43%) per test for the GO and $297.9J$ (a gain of 40.9%) for the Swiftkey. Regarding execution time, the tests took an average of 243 seconds less for the GO and 222.2 seconds less for the Swiftkey.

4 STUDIES DISCUSSION

In this section we present the analysis and discussion for each of the studies described previously. For the automated experiment we analyze the results obtained for each one of the testing modes (with *minimal* and *default* features). We describe and provide details of the statistical evidences used to retrieve meaningful conclusions regarding the testing modes and individual keyboards energy consumption. Considering the real user experiment, we evaluate the gains associated of word prediction features and the impact of some aspects intrinsic to each user on energy consumption. We also compare the procedure executed with real users with the naive automatic version, described in Section 2.3.2.

By examining the data illustrated in Figure 2, it is possible to observe that there are significant differences among the energy consumption of the different KUT's. For example, looking at the median values in both testing modes, and comparing the most efficient keyboard with the least efficient, it is safe to conclude that there are significant gains. For instance, in the default mode, the GO keyboard has an efficiency gain of more than 18% compared to the Google keyboard. In the minimal mode, the gain of the more energy efficient keyboard (Cheetah) compared to the more energy-greedy (GO) is lower (8.3%).

The illustrated data gives us a clear indication on what keyboard behaved as the most energy friendly for each scenario. Nevertheless, it is important to prove if there is statistical evidence behind these observations, that is, if the energy consumed by each keyboards is statistically different from the consumption of the others. Thus, we tested the following hypothesis, for both default and minimal modes:

> $\boldsymbol{H}_0$: The energy values obtained for each KUT follow the same distribution;
>
> $\boldsymbol{H}_1$: The energy values obtained for each KUT does not follow the same distribution;

The evaluation of these hypothesis and consequently rejection of the null hypothesis H_0 (allows us to answer to **RQ1**. If a statistical test indicates that two datasets follow different distributions, and we can see a clear tendency favoring one of them, we can conclude that indeed there are evidences that there are differences in

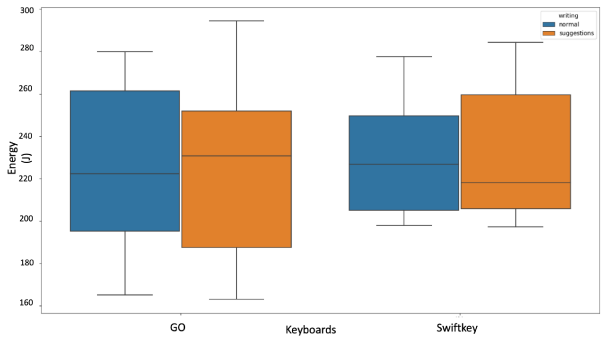

Figure 5: Energy consumption box plots of the 2 keyboards tested with real users

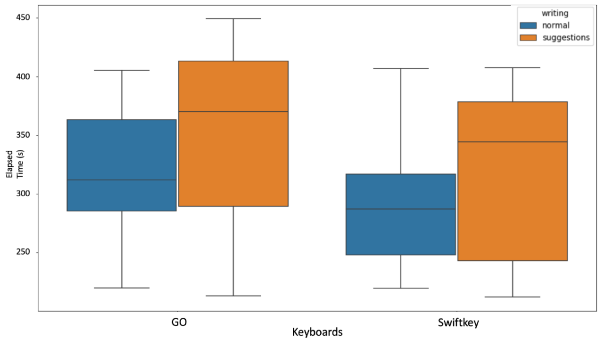

Figure 6: Tests Elapsed time of both keyboards tested with real users

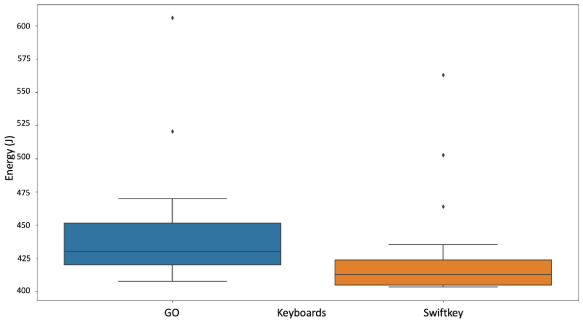

Figure 7: Energy consumption box plots of the 2 keyboards for the naive automatic procedure

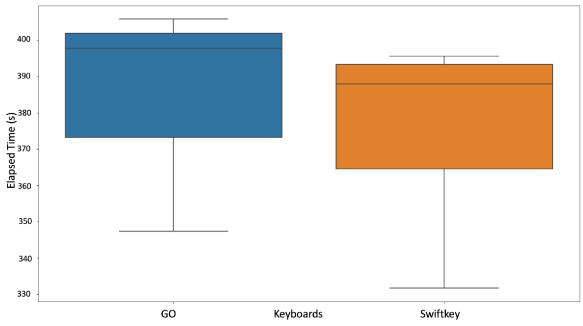

Figure 8: Tests Elapsed time of both keyboards for the naive automatic procedure

energy consumption between the KUT's. To understand if there is an overall significant relevance between the distributions of the KUT's energy values, first, we analyzed each one of the datasets of each keyboard, to evaluate whether these groups are parametric or not. Since only one keyboard presented values followed a Gaussian distribution, we assumed a non-parametric behavior of the groups and ran the Kruskal Wallis H test [17].

The Kruskal Wallis test can be used to determine whether more than two independent non-parametric samples have a different distribution. We ran this test twice, For each keyboard mode, we evaluated the p-value considering $\alpha = 0.05$. The p-values of 0 and 0.026 for the KUT's samples in default and minimal mode, respectively, obtained for this test, allow us to reject $\boldsymbol{H}_0$, and respond to **RQ1**, concluding that indeed there are statistical differences in energy consumption between the selected keyboards, in both modes. This result can be explained with the fact that these keyboards are complex applications and that differ in many aspect, from layout aspects, animations or word suggestion algorithms. Consequently, different characteristics and feature implementations might have different impacts on energy consumption.

As described in the previous section 3, in the case of GO and Swiftkey, these keyboards consumed more energy in the minimal mode than in the default mode. GO is the most evident case, with a median value 10.2% higher. Swiftkey, however, registered a less significant difference, of only 3.8%. The other keyboards decreased their energy consumption in the minimal modes. The most significant case is the Cheetah, with a decrease of 9.3%. This may be due to the fact that this keyboard uses a greater number of animations by default, such as key pressing animations or an animated background. Since these animations demand additional computation to be rendered, this sharp descent can be justified by the absence of these animations in minimal mode.

In order to assess whether or not there is significant and statistically supported difference between using the different modes, we need to perform another statistical test. A suitable test for this purpose is Mann-Whitney U test [19], since is capable to evaluate whether two independent samples were selected from populations having the same distribution. This test was performed for each keyboard, in which the following hypotheses were evaluated:

> $\boldsymbol{H}_0$: The energy values obtained for both modes follow the same distribution;
> $\boldsymbol{H}_1$: The energy values obtained for both modes doesn't follow the same distribution;

For the non-parametric groups (all except GO), we used the Mann-Whitney U test [19], considering $\alpha = 0.05$. The results obtained are detailed in table 4. The only keyboard that doesn't have significant statistical differences is Google, with Cheetah being the strongest null hypothesis reject. For the Go keyboard, a different statistical test was performed, as its samples follow a normal distribution, according to 2 different tests performed (Shapiro [30]

and D'Agostino and Pearson's [14]). In order to evaluate the same hypothesis, for parametric data, the alternative is the T-test [32], a two-sided test for the null hypothesis that 2 independent samples have identical expected values. For the same value of α, we obtained a p-value o 0., that also allows to reject H0.

Keyboard	p-value	Decision
Cheetah	0	Reject H0
Google	0.076	Failed to reject H0
Fancykey	0	Reject H0
Swiftkey	0.036	Reject H0

Table 4: p-values and hypothesis test decision for each of the KUT with non-parametric samples

The results obtained for both parametric and non-parametric samples, combined with the comparison of the values presented in the plots from Figure 2, allow us to answer **RQ2**: there are in fact cases where it is possible to save energy by disabling keyboard features. We observed that for Cheetah, Google and Fancykey. Go and Swiftkey surprisingly do not reflect this behavior. A possible justification for this may be that the developers were more concerned with optimizing the default activated features, or even that such features contain performance improvements that are removed with a feature being disabled.

The data from the correlation matrix in Figure 4 help to understand the correlation of the metrics surveyed during the test execution process. With regard to energy consumption, there are expected correlation values, such as CPU frequency, and power, since Android multiplies the CPU time for each application by the voltage required to run the CPU at a specific frequency [7]. We also can observe that energy consumption of the keyboards decreased along the 25 tests (each test was identified by its index, column 'testid'), given the slightly red value present in the figure, as well as the elapsed time of the tests. This observation is consistent with a previous study that shows that less execution time does not necessarily indicate less energy consumption [15].

As for the results obtained with the use of suggestions, the results were somewhat unexpected in the study with real users. The word prediction and suggestion mechanisms were developed in order to save keyboard users time and effort when writing text. However, the results obtained with the 20 users who used the keyboards trying to maximize the use of suggestions ended up going against this assumption, as they took longer to write the requested text (figure 6. for the median value of each group, the Go keyboard had an increase of 16.6% and the Swiftkey of 15.7%. In the case of GO, it ended up consuming more energy, but the opposite happened for the Swiftkey (which goes against the results illustrated in the table 2, in which it consumed more energy when fewer features were used (figure 5. Taking into account these data, the answer to the question **RQ3** would be negative, however, during the execution of the tests with the real users, it was observed that they always showed great concern in actually observing at each letter inserted if the suggestion appeared in the respective bar. Also they didn't recognize the text that they were typing and this study would probably have different results if they were writing a text they knew or had previously thought about.

Nevertheless, in the automatic naive approach performed to simulate the same procedure, but without resorting to real users, it obtained concordant results with the assumption that the use of suggestions can increase writing efficiency, both in terms of energy and time. Since the default mode of the first study correspond to the automatic version of the mode that minimized the suggestions in the studies with real users, we can compare those results with the ones with the this naive procedure. By analyzing the results of this procedure, described in figures 7 and 8, we can observe that the median of the values obtained for both keyboards using suggestions is significantly lower in terms of energy and time than the ones obtained in the first study in default mode (figure 2 and 3. More specifically, for the median values obtained for each of the keyboards, a gain of 311.74 Joules(43%) per test for the GO and 297.91 (40.9%) for the Swiftkey.

Given the results obtained with the real users and the naive approach, the answer to **RQ3** is: Sometimes. The study with real users has controversial results, where the use of word suggestions had negative impact in energy consumption and elapsed time. However, the same study realized under a different scenario, like writing 100 random words of user choice, could have different results than the ones obtained with the transcription of the selected text. Nevertheless, the results off the naive automatic approach satisfy the assumption that in fact, the use of this kind of features can help users to save energy and time while typing text in virtual keyboards.

5 THREATS TO VALIDITY

Android is a multi-process operating system, where multiple processes and applications run "simultaneously". To ensure that during the execution of each test, only the intended application is running on the device, affecting its consumption, we registered its state before and after each test. Moreover, we executed the automatic tests in a customized boot images (using root-privileges) where such processes were disable. Testing in customized rooted images is an usual approach followed by research community [4, 18, 21, 22]. The device selected for this work is usually used by the research community to conduct performance studies [5, 29] due to the fact that does not have many pre-installed 3rd party-apps. Moreover, we run Android version 6.0.1, that uses the ART virtual machine. However, there are several compiling options available for this version for applications and system source code, and for 3rd party-apps, by default this version uses the quicken option, that only optimize some android bytecode instructions to achieve better interpreter performance. Furthermore, to minimize eventual optimizations that the system might perform for both AUT and profiler code, we restore the boot image between sets of tests for each keyboard. Thus, in the second study, we also wipe the Dalvik/ART and system cache, in order to delete eventual optimizations and execution profiles provided by JIT compilation. To minimize external interference, the automated tests for the default and minimal versions were performed with the minimum value of brightness in order to minimize the impact on power consumption. For the user

[7] Android Power profiles:https://source.android.com/devices/tech/power

study we used a level of 50% of brightness so that users could see the screen.

As for factors intrinsic to the KUT, in order to establish fair and adequate comparisons among keyboards, some precautions had to be taken. All keyboards used the same language (EN-US),the language of the system in which they were tested and the text to be inserted, since the chosen language has implications for the keyboard layout and dictionaries. Some keyboards also have some peculiarities with which we had to be cautious and ascertain their impact. For instance, Go Keyboard offers a memory boost option, enabled by default. When tested in minimal mode, we evaluated the eventual gains in using this option, but the energy values obtained were slightly higher than the ones obtained with this option turned off. So we ended up not using it in the minimal testing mode.

As for the results obtained, these may differ between devices, Android versions or the keyboard application. Regarding the automatic procedure performed, there was a noticeable delay in the communication between the workstation and the Android device, caused by the type of interface (ADB), communication protocol(USB 3.0) and by the system version. In a test carried out on newer devices and emulators, we found that the typing rate using the same automatic procedure was substantially higher, which can eventually lead to different results. However, we wanted to carry out the study on the currently most used version of the platform.

Regarding the study performed with real users, the chosen case studies (maximizing and minimizing the use of suggestions in the transcription of a text) may not have been the most suitable to assess the advantage of using predictions. The language used for keyboard and text dictionaries to be entered may also have implications. Although all participant who wrote the text learned the language since elementary school, the results could possibly be more satisfactory if the language chosen was their native language. The process of transcribing words and the indication that they should not make mistakes in writing meant that there was a delay in reading the text to be transcribed and verifying that the word they had just written was right. This delay would probably be inferior if they know the text before-hand and/or did not had to transcribe it.

6 RELATED WORK

When it comes to measuring energy consumption on Android, there are currently several studies to offer powerful alternatives to estimate or measure energy consumption. Like Trepn, there are other software-based tools capable of assessing the power consumption of the system, applications and portions of source code. Having only the application APK and no source code, GreenScaler [6] can be used to estimate energy consumption of an application execution. Having access to the source code, Petra [22] is an alternative to consider to measure and locate the energy consumption of applications. GreenScaler is an approach described as model-based, which uses data from previous work by the same authors [16] to calibrate the model. PETRA has emerged with an approach using APIs available since the version 5 of Android[8]. The authors claim that this tool can estimate the energy consumption of the source code of an Android application with a low granularity, at the method level, providing accurate consumption results [23].

There is a small set of works in the literature with the aim of looking at the performance of virtual keyboards. The first work to look at virtual keyboards from an power usage point of view was [24] in 2012, where the authors analyzed the power consumption profile of text input methods on smartphones. They considered 3 different keyboards, where one of them counts currently with only 1000 downloads and another no longer exists. Given the evolution recorded on the platform, on current smartphones and on the platform's keyboards, these results may be out of date. In addition, this work did not consider the impact of intrinsic features on keyboards or studies with real users. More recently, there is a work [31] focused on assessing intrinsic aspects of keyboards and the screen sizes of devices. More specifically, it intended to assess the effects of keyboard size, gap and button shape on usability metrics in thumb interaction in mobile touchscreen devices.

Regarding the evaluation and benchmarking of different applications and development tools the same effect on the Android platform, Wilke et al [33] evaluated the energy performance of different browsers and email clients for the android platform. Corbalan et al [7] evaluated the difference in energy consumption of applications developed with different development frameworks for 3 different tasks: intensive processing of video and audio reproduction.

7 CONCLUSIONS AND FUTURE WORK

In this work we have explored the performance of virtual keyboards in the Android platform, focusing on their impact on energy consumption. We considered 5 widely used keyboard applications and designed an experiment to evaluate their energy consumption, in different execution scenarios. We performed different studies using both real users and automatic procedures to simulate real user interaction. The results from this experiment show that indeed there are differences in the energy consumed by the selected keyboards, and switching keyboards or enabling/disabling settings promotes energy saving in most cases. In fact, replacing the most energy greedy keyboard by the greenest one has reduced energy consumption in 18%, and when advanced features of those keyboards (such as word prediction or animations) are turned off, the energy consumption is also reduced, in this case up to 9.3%.

As future work, we intend to conduct a larger study, both in terms of users and devices, to evaluate the individual impact that each of the keyboards features may have on energy consumption and the reason of such impact. The study will allow us to determine the best keyboard and the best combination of features that optimize energy consumption and, if possible, according to users' usability preferences. Android is a fast evolving ecosystems, and the impact on virtual keyboards of mechanisms like compilation schemes, virtual machines and garbage collection should also be considered for analysis, in order to determine the greenest keyboards for a particular combination of mobile device and system version. Thus, an automated technique to evaluate the energy of keyboards needs to be considered and the algorithms we presented in Section 2 is a step in this direction.

[8] https://developer.android.com/about/versions/android-5.0.html#Power

REFERENCES

[1] 2020. Distribution dashboard. https://developer.android.com/about/dashboards

[2] 2020. Monkeyrunner. https://developer.android.com/studio/test/monkeyrunner

[3] A. R. Bakker. 2014. Comparing Energy Profilers for Android. In *Proceedings of 21st Twente student conference on IT, Enschede, The Netherlands*.

[4] A. Banerjee and A. Roychoudhury. 2017. Future of Mobile Software for Smartphones and Drones: Energy and Performance. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 1–12. https://doi.org/10.1109/MOBILESoft.2017.1

[5] Angel Cañete, Jose-Miguel Horcas, Inmaculada Ayala, and Lidia Fuentes. 2020. Energy efficient adaptation engines for android applications. *Information and Software Technology* 118 (2020), 106220. https://doi.org/10.1016/j.infsof.2019.106220

[6] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2019. GreenScaler: Training Software Energy Models with Automatic Test Generation. *Empirical Softw. Engg.* 24, 4 (Aug. 2019), 1649–1692. https://doi.org/10.1007/s10664-018-9640-7

[7] Leonardo Corbalan, Juan Fernandez, Alfonso Cuitiño, Lisandro Delia, Germán Cáseres, Pablo Thomas, and Patricia Pesado. 2018. Development Frameworks for Mobile Devices: A Comparative Study about Energy Consumption. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 191–201. https://doi.org/10.1145/3197231.3197242

[8] Leonardo Corbalan, Juan Fernandez Sosa, Alfonso Cuitiño, Lisandro Delia, Germán Cáseres, Pablo Thomas, and Patricia Pesado. 2018. Development frameworks for mobile devices: a comparative study about energy consumption. 191–201. https://doi.org/10.1145/3197231.3197242

[9] Marco Couto, Tiago Carção, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2014. Detecting Anomalous Energy Consumption in Android Applications. In *Programming Languages*, Fernando Magno Quintão Pereira (Ed.). Springer International Publishing, 77–91.

[10] L. Cruz and R. Abreu. 2017. Performance-Based Guidelines for Energy Efficient Mobile Applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 46–57. https://doi.org/10.1109/MOBILESoft.2017.19

[11] Luis Cruz and Rui Abreu. 2018. Measuring the Energy Footprint of Mobile Testing Frameworks. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. ACM, New York, NY, USA, 400–401. https://doi.org/10.1145/3183440.3195027

[12] Luis Cruz and Rui Abreu. 2018. Using Automatic Refactoring to Improve Energy Efficiency of Android Apps. *CoRR* abs/1803.05889 (2018). arXiv:1803.05889 http://arxiv.org/abs/1803.05889

[13] L. Cruz, R. Abreu, and J. Rouvignac. 2017. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 205–206. https://doi.org/10.1109/MOBILESoft.2017.21

[14] Ralph B. D'Agostino. 1971. An omnibus test of normality for moderate and large size samples.

[15] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. 2013. Estimating mobile application energy consumption using program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 92–101. https://doi.org/10.1109/ICSE.2013.6606555

[16] Abram Hindle. 2013. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering* 20 (04 2013). https://doi.org/10.1007/s10664-013-9276-6

[17] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621. https://doi.org/10.1080/01621459.1952.10483441 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/01621459.1952.10483441

[18] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/2597073.2597085

[19] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 18, 1 (03 1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[20] Hugo Matalonga, Bruno Cabral, Fernando Castor, Marco Couto, Rui Pereira, Simão Melo de Sousa, and João Paulo Fernandes. 2019. GreenHub Farmer: Real-World Data for Android Energy Mining. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, 171–175. https://doi.org/10.1109/MSR.2019.00034

[21] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. 2018. EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. *IEEE Transactions on Software Engineering* 44, 12 (Dec 2018), 1176–1206. https://doi.org/10.1109/TSE.2017.2757486

[22] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia. 2017. PETrA: A Software-Based Tool for Estimating the Energy Profile of Android Applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 3–6. https://doi.org/10.1109/ICSE-C.2017.18

[23] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia. 2017. Software-based energy profiling of Android apps: Simple, efficient and reliable?. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 103–114. https://doi.org/10.1109/SANER.2017.7884613

[24] Henry Obison and Chiagozie Ajuorah. 2013. Energy Consumptions of Text Input Methods on Smartphones.

[25] Gustavo Pinto and Fernando Castor. 2017. Energy Efficiency: A New Concern for Application Software Developers. *Commun. ACM* 60, 12 (Nov. 2017), 68–75. https://doi.org/10.1145/3154384

[26] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining Questions About Software Energy Consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 22–31. https://doi.org/10.1145/2597073.2597110

[27] J.K Rowling. 2000. *Harry Potter and the Goblet of Fire* (1 ed.). Scholastic.

[28] Rui Rua, Marco Couto, Adriano Pinto, Jácome Cunha, and João Saraiva. 2019. Towards using Memoization for Saving Energy in Android. In *CIbSE*.

[29] R. Rua, M. Couto, and J. Saraiva. 2019. GreenSource: A Large-Scale Collection of Android Code, Tests and Energy Metrics. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 176–180. https://doi.org/10.1109/MSR.2019.00035

[30] S. S. Shapiro and M. B. Wilk. 1965. An analysis of variance test for normality (complete samples)†. *Biometrika* 52, 3-4 (12 1965), 591–611. https://doi.org/10.1093/biomet/52.3-4.591 arXiv:https://academic.oup.com/biomet/article-pdf/52/3-4/591/962907/52-3-4-591.pdf

[31] Da Tao, Tieyan Wang, Haibo Tan, Jian Cai, and Xu Zhang. 2020. Understanding One-Handed Thumb Interaction with a Mobile Touchscreen Device: Effects of Keyboard Size, Gap and Button Shape. In *Advances in Usability and User Experience*, Tareq Ahram and Christianne Falcão (Eds.). Springer International Publishing, Cham, 412–423.

[32] B. L. Welch. 1947. The Generalization of 'Student's' Problem When Several Different Population Variances are Involved. *Biometrika* 34, 1-2 (01 1947), 28–35. https://doi.org/10.1093/biomet/34.1-2.28 arXiv:https://academic.oup.com/biomet/article-pdf/34/1-2/28/553093/34-1-2-28.pdf

[33] Claas Wilke, Christian Piechnick, Sebastian Richly, Georg Püschel, Sebastian Götz, and Uwe Aundefinedmann. 2013. Comparing Mobile Applications' Energy Consumption. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. Association for Computing Machinery, New York, NY, USA, 1177–1179. https://doi.org/10.1145/2480362.2480583