A Framework for the Automatic Execution of Measurement-based Experiments on Android Devices

Ivano Malavolta[1], Eoin Martino Grua[1], Cheng-Yu Lam[1], Randy de Vries[1], Franky Tan[1], Eric Zielinski[1], Michael Peters[2], Luuk Kaandorp[1]
[1] Vrije Universiteit Amsterdam, The Netherlands. i.malavolta@vu.nl, e.m.grua@vu.nl, c2.lam@student.vu.nl, randy.de.vries@student.vu.nl, k.h.tan@vu.nl, e.a.zielinski@student.vu.nl, l.kaandorp@student.vu.nl
[2] M2mobi, The Netherlands. m.peters@m2mobi.com

ABSTRACT

Conducting measurement-based experiments is fundamental for assessing the quality of Android apps in terms of, *e.g.,* energy consumption, CPU, and memory usage. However, orchestrating such experiments is not trivial as it requires large boilerplate code, careful setup of measurement tools, and the adoption of various empirical best practices scattered across the literature. All together, those factors are slowing down the scientific advancement and harming experiments' replicability in the mobile software engineering area.

In this paper we present Android Runner (AR), a framework for automatically executing measurement-based experiments on native and web apps running on Android devices. In AR, an experiment is defined once in a descriptive fashion, and then its execution is fully automatic, customizable, and replicable. AR is implemented in Python and it can be extended with third-party profilers.

AR has been used in more than 25 scientific studies primarily targeting performance and energy efficiency.

1 INTRODUCTION

Android is the leading mobile platform today and the majority of scientific contributions on mobile software engineering focuses on Android [1]. When dealing with quality properties like energy efficiency and performance, practitioners and researchers rely on the measurement of run-time metrics such as battery discharge, CPU and memory usage, number and type of network requests, etc. [7, 9, 10] In this context, considerable effort and time are spent on setting up infrastructures and software pipelines for conducting measurement-based experiments. Moreover, when available, existing pipelines are either (i) ad-hoc for a specific experiment or (ii) tailored to one specific quality property (*e.g.,* energy consumption).

This paper presents Android Runner (AR)[1], a framework to automatically execute measurement-based experiments on native and web apps. The main goal of AR is to *streamline the execution of measurement-based experiments involving Android devices*. In AR, experiments are defined in a descriptive fashion, and then their execution is fully *automatic*, *customizable*, and *replicable*. We designed AR with the following design drivers in mind:

- **Automation**: after an initial configuration, the experiment can be executed without any interaction from the user;
- **Incremental experiments**: AR always persists the intermediate results of the experiment and, if interrupted, it is able to resume it and continue with the remaining runs;
- **Usability**: users define the experiment in a *descriptive* manner, without writing boilerplate code or knowing the internals of AR;
- **Customizability**: users have the possibility to include their own business logic and automated testing tools [6] at specific points within the experiment execution (*e.g.,* before the whole experiment begins, before or after each run, etc.);
- **Profiler independence**: in AR, run-time measures can be collected both via hardware (*e.g.,* the Monsoon power monitor[2]) and software (*e.g.,* Trepn[3]). Profilers can produce different data points and can interact with apps and the Android device in their own way; moreover, AR makes straightforward to use multiple profilers within a single experiment;
- **Experiments replicability**: given the configuration, subject apps, and available Android devices, AR can execute an already-performed experiment with low effort, even if the experiment has been performed by a third party.

We are aware that frameworks like AR must be as accurate as possible and that their accuracy must be independently verifiable. In order to facilitate the validation of AR, we created a set of 27 **benchmarking apps**, each of them stressing a specific hardware component of an Android mobile device, such as its accelerometer, camera, CPU, display, GPS, etc.. We use those apps on a regular basis for validating the accuracy of AR. The full set of benchmarking apps is publicly available[4] (together with their source code) and can be reused by researchers in the area of mobile software engineering, also independently of AR.

The **target audience** of AR includes (i) researchers who need to conduct empirical evaluations of software engineering methods and techniques involving Android apps, (ii) researchers developing new run-time profilers for Android devices, and (iii) practitioners needing to quantitatively assess the quality of their own apps.

The remainder of the paper is organized as follows. Section 2 provides an overview of the proposed framework, Section 3 presents its

[1] https://github.com/S2-group/android-runner

ASEW '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-8128-4/20/09…$15.00
https://doi.org/10.1145/3417113.3422184

[2] https://www.msoon.com/high-voltage-power-monitor
[3] https://developer.qualcomm.com/forums/software/trepn-power-profiler
[4] https://github.com/S2-group/android-apps-benchmark

currently implemented plugins, and Section 4 describes its intended usage. Section 5 presents the set of synthetic apps we developed for benchmarking AR, together with an application in the area of energy efficiency. Section 6 closes the paper.

2 OVERVIEW OF THE FRAMEWORK

In the remainder of this section we describe the input and output artifacts of AR (Section 2.1) and its architecture (Section 2.2).

2.1 Input and Output

As shown in Figure 1, AR takes four types of artifacts as input.

Devices. It is a configuration file mapping the Android Debug Bridge (ADB[5]) identifier of each connected device to a mnemonic name. This makes the definition of the experiment independent of the used mobile devices, thus allowing its direct replication on different devices.

Experiment configuration. It is the main input of AR and contains the description of the experiment. Table 1 describes its main fields. The full list of the fields supported by AR is available in its GitHub repository.

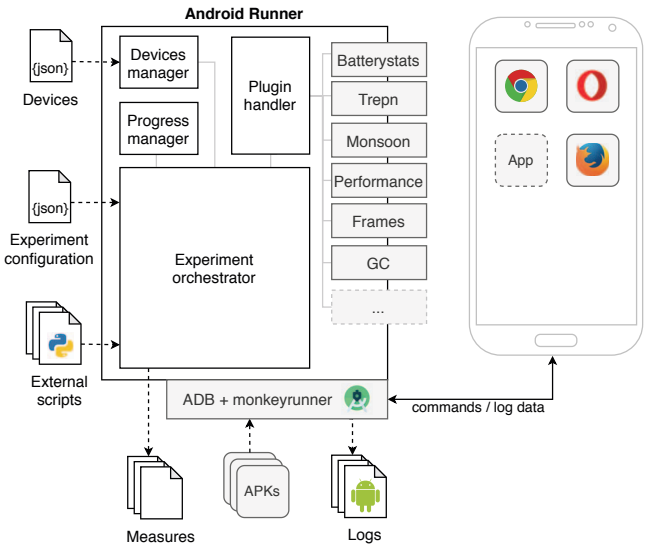

Figure 1: Overview of AR

External scripts. A set of optional Python scripts provided by the user for executing custom behaviour during the execution of the experiment. AR passes contextual information to external scripts, such as the current subject of the experiment, the device on which the subject is running, etc. AR supports the following hooks for executing custom scripts:

- `before_experiment`: executes before the first run of the experiment;
- `before_run`: executes before every run of the experiment;
- `after_launch`: executes after the current subject is launched, but before measurement starts;
- `interaction`: executes during the measurement window (it is often used for executing usage scenarios like user taps, drags, etc.); for convenience, instead of Python, the user can also adopt a JSON-based descriptive syntax where user interactions are specified as sequences of typed actions with display coordinates and timestamps. Listing 1 shows an example with a touch, drag, and typing actions. In this way, complex usage scenarios can be automatically recorded once and AR is able to replay them in each run using the `monkeyrunner`[6] Android tool;
- `before_close`: executes before the current subject is closed;
- `after_run`: executes after a run completes;
- `after_experiment`: executes after the last run of the experiment.

```
1 {"type": "touch", "x": 600, "y": 200, "down": 200,
      "up": 300}
2 {"type": "drag", "points": [{"x": 600, "y": 200},
      {"x": 600, "y": 800}] "down": 2200, "up": 2300}
3 {"type": "press", "keys": [{"key": "KEYCODE_BACK"}],
      "down": 3200, "up": 3300}
```

Listing 1: Simple usage scenario with three actions

Over time, external scripts proved to be useful in many situations, *e.g.*, for setting up a local web proxy recording the traffic generated by the subjects, for instrumenting the subjects of the experiment on the fly, or for cleaning up the environment between runs.

APKs. The binary files of the subject apps of `native` experiments. AR will automatically fetch, install, execute, and uninstall them.

AR produces two types of outputs: measures and logs.

Measures. The measures produced by the profilers (*e.g.*, consumed energy). The specific format of the measures is profiler-dependent and is decided by the developers of the plugin of each profiler according to their specific needs/constraints. So far, the vast majority of implemented plugins produce comma-separated values (CSV) files (see Section 3). For the sake of replicability and independent verification, we require that the profilers used in AR must always save their results into the file system of the user.

Logs. The dump of all system messages produced during the whole experiment. Before each run, AR launches the `logcat`[7] Android debugging tool, receives its messages as a stream, and persists them for future inspection. Log files have proven to be useful for performing diagnostic checks on problematic runs of the experiment and for further inspecting low-level system information such as calls to the garbage collector, invocations of system APIs, etc.

2.2 Architecture

In line with the state of the art [3–5, 10], AR experiments involve two main hardware nodes: a *base station* running the software in charge of executing the experiment (AR in our case) and a *mobile device* running the subjects. The main rationale for having two separate nodes is to keep the Android device as lightweight as possible, so as to not influence the measurements [10]. AR is implemented as a set of Python 3 modules and as such it can run on any base station able to run Python code, such as a desktop computer, a laptop, or a Raspberry Pi; given its low cost, the latter is especially useful when needing to run multiple experiments in parallel.

[5] https://developer.android.com/studio/command-line/adb

[6] https://developer.android.com/studio/test/monkeyrunner

[7] https://developer.android.com/studio/command-line/logcat

Table 1: Main fields of the experiment configuration

Name (type)	Description
Type (native | web)	– native : the experiment subjects are native apps; they are identified either in the paths or apps fields. – web : the subjects of the experiment are mobile-optimized web apps; in this case, the paths field will contain their URLs and the browser field must be set.
Randomization (boolean)	True if the order of execution of experiment runs is randomized, false otherwise.
Repetitions (number)	The number of times an experiment trial is repeated; it is useful for taking into consideration the natural fluctuations when dealing with measures like execution time, energy consumption, memory usage, etc.
Time between runs (number)	Waiting time between each run of the experiment; it is useful for mitigating the effect where certain hardware components of mobile devices are optimistically kept active by the OS to avoid startup energy costs [9].
Devices (map)	Contains an entry for each Android device to be used in the experiment (identified by its name in the `Devices` file), together with other device-specific information (*e.g.*,, whether it is rooted or not).
Paths (list)	The list of either the file system paths of the subjects (if native) or their URLs (if web).
Apps (list)	The package names of the subject apps in case they are already installed on the device; it is useful for system-level preinstalled apps for which there is no need in having a locally-stored APK.
Browsers (list)	The browsers to use when running an experiment of type web . Currently AR supports the following browsers: Google Chrome, Mozilla Firefox, Opera. Different browsers can be used within the same experiment.
Profilers (map)	Contains an entry for each profiler used for collecting run-time measures during the experiment. Since AR is profiler-independent, here each entry corresponds to one of the external plugins shown in Figure 1 (*e.g.,* Batterystats, Trepn, Monsoon) and contains plugin-specific parameters, such as sampling frequency, data points to collect, etc.
Scripts (map)	Optional, contains an entry for each external script implementing custom behaviour (see details below).
Duration (number)	Optional, AR will terminate the execution of a run if it does not terminate within the provided duration in ms.

AR is independent from the communication channel between the base station and the Android device (*e.g.,* USB, Wi-Fi), provided that it is supported by ADB. ADB is the official Android tool to run commands on a connected Android device. In experiments targeting energy consumption, it is advised to use the Wi-Fi channel in order to avoid the influence that USB charging may have on the energy-related behaviour of the device. Alternatively, for rooted devices AR is able to overcome this limitation by programmatically disabling the USB charging of the device while running the experiment.

As shown in Figure 1, the main component of AR is the **Experiment orchestrator**. It is in charge of executing the whole experiment according to the experiment configuration provided by the user. The orchestrator performs the following steps:

(1) Perform *diagnostic* checks on user input (*e.g.,* all subjects' APKs are present) and the environment (*e.g.,* check if external tools like ADB are properly configured);
(2) Setup the *devices* via the Devices manager;
(3) Bring up the *profilers* specified in the experiment configuration via the Plugin handler;
(4) Setup the *plan of the runs* via the Progress manager;
(5) *Execute the runs* according to the sequence defined in the plan. For each run the orchestrator does the following steps:
  (a) if the experiment is native and the current subject is not installed on the device, install it;
  (b) if the experiment is web , launch one of the browsers according to the plan and wait for its full load;
  (c) start all profilers (via the Plugin handler);
  (d) launch the subject, *i.e.,* launch the app if the experiment is native or load the web app in the browser if it is web ;
  (e) stop all profilers (via the Plugin handler);
  (f) close and clear the subject, *i.e.,* terminate the app (or the web browser) and delete all data associated with it (*e.g.,* the cache of the browser, the local database of the app);
  (g) aggregate and persist the collected measures if (i) the plugin implements this feature (see below) or (ii) the user provided a run-specific aggregation script;
  (h) persist all logs via `logcat`;
  (i) inform the Progress manager that the run is completed.
(6) *Clean up* the devices via the Devices manager (*e.g.,* uninstall all subjects in case of native experiment);
(7) *Aggregate the collected measures* according to either plugin-specific or (if present) user-defined aggregation scripts.

During the execution of these steps, AR executes the external scripts provided by the user according to the specific hooks they are linked to (*e.g.,* before/after each run, etc.).

The **Devices manager** is responsible for providing a layer of abstraction on the low-level operations involving the Android devices. Specifically, it is able to (i) check if the devices specified by the user in the input configuration files are usable in the experiment, (ii) programmatically activate/deactivate the USB battery charging functionality during the experiment execution (this applies only to rooted devices), (iii) list, install, and uninstall apps on the devices, (iv) launch, stop, and force-terminate processes on the device, (v) read and write files on the devices, etc.

In addition to masking the complexity of low-level operations on the devices, the Devices manager also allows us to localize the impact of future changes in AR in case new non-backward-compatible versions of ADB will be released or for experimenting with completely new platforms in the future.

The main responsibility of the **Progress manager** is to keep track of the execution of each run of the experiment. Its first operation is to build the *plan* of all the runs of the experiment; each entry of the plan is a tuple composed of (i) a unique identifier of the run, (ii) the subject to be executed, (iii) the device on which the subject must be executed, and (iv) the index of the current run within the range from 0 up to the number of *repetitions* per trial specified by the user (see Table 1). Then, if the user requires *randomization*, the entries of the plan are shuffled. Finally, the resulting plan is sent to the Experiment orchestrator, which will go through each entry of the plan, executes it, and informs the Progress manager about the completion of the run. During the experiment execution, the Progress manager persists the current entry of the plan being executed and additional metadata; at launch time, this information is queried by the Experiment orchestrator so that partially-executed experiments can be safely resumed. This feature is particularly useful when large-scale experiments must be carried out in *batches* or when an unexpected error occurs and the experiment execution must be restored without incurring in data loss.

The **Plugin handler** provides (i) a set of facilities for managing the profilers and (ii) an extension point that third-party developers can use for integrating their own measurement tools into AR. When launching AR, the Plugin handler checks which plugins are currently extending its extension point and makes their functionalities available to the user. The extension point exposes a common Python API to plugin developers, so that they can interact with the other components of AR. For example, it allows developers to access information about the current subject of the experiment, to interact with ADB, to access the information provided by the Devices manager, etc.

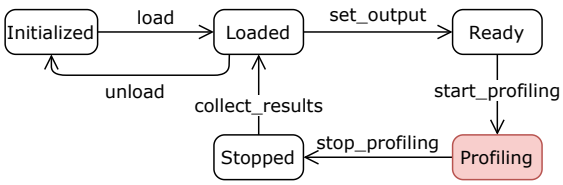

Figure 2: Plugin lifecycle

Each plugin follows the lifecycle in Figure 2 to allow the Experiment orchestrator to suitably call it during the execution of the experiment and to ensure that multiple plugins can be used during the same experiment. In the *initialized* state the plugin passes an initial check of its configuration (*e.g.,* the configuration of the Monsoon power monitor is valid), then the plugin is *loaded* (*e.g.,* an energy profiling app like Trepn is installed on the device). The plugin reaches the *ready* state when its output folder exists and is writable; when the profiling is started by the orchestrator, the plugin reaches the *profiling* state and continuously collects measures until it is *stopped*. Finally, the resulting measures are collected by the orchestrator and, if other runs need to be executed, the plugin comes back to the *loaded* state, otherwise it is unloaded.

A plugin can provide its collected measures either raw or aggregated. Each plugin can define its default aggregation policy, either for each run or after the whole experiment is completed. In the first case the aggregation happens when the `collect_result` event is triggered, whereas in the second case it happens when the `unload` event is triggered. Optionally, users can also specify their own experiment-specific aggregation policies and override the ones provided by the plugins. In this way, plugin developers can simply focus on producing the measures and providing a default aggregation policy, instead of trying to cover all possible use cases for their plugin.

3 IMPLEMENTED PLUGINS AND RELATION WITH RELATED WORK

With AR we do not want to compete with already-existing profilers or measurement frameworks like GreenMiner [5] or PETrA [3], but rather we aim at providing a solid, verifiable, and replicable backbone to which already-existing profilers can be integrated. To this purpose, AR follows a plugin-based architecture where third-party profilers can be developed and integrated independently of the orchestration logic of AR. The intuition behind this design decision is to avoid researchers and practitioners to reinvent the wheel with ad-hoc pipelines, but rather to focus their efforts on their specific measurement needs by developing AR plugins. Table 2 lists the currently available AR plugins.

Table 2: Available AR plugins

Name (quality)	Description
batterystats (Energy)	Uses the `batterystats` utility and estimates energy consumption via the algorithm proposed in [3].
monsoon (Energy)	Collects energy consumption via the Monsoon hardware profiler and the Physalia[8] tool.
trepn (mixed)	Collects data via the Trepn profiler, *e.g.,* power consumption, battery temperature, CPUs frequency
mem-CPU (Perf.)	Collects memory and CPU usage via the `cpuinfo` and `meminfo` Android utilities.
frametimes (Perf.)	Collects frame rendering durations and the number of delayed frames with the technique used in [4].
gc (Perf.)	Collects the number of garbage collections as in [4].
perfume-js (Perf.)	Collects performance metrics via the Perfume.js library[9], like time to first byte and first paint.

It is important to highlight that ARis not limited to energy and performance measurement; it can be extended to support many kinds of measurement-based studies related to Android apps, provided that a suitable plugin supporting the required measures exists (*e.g.,* one monitoring at run-time calls to sensitive Android APIs).

4 INTENDED USAGE OF AR

At the time of writing, more than 25 studies are based on AR, ranging from peer-reviewed publications, theses, and students' projects. For the sake of space, below we report the ones that are representative of the intended usage scenarios of AR.

Web experiments. In [8] we used AR and its *batterystats* plugin for empirically assessing the impact of the cache of Progressive Web Apps (PWAs) on their energy consumption and page load time. The experiment involves one smartphone running Android 9.0, 9 PWAs locally served by the base station over Wifi, the Mozilla Firefox as

[9]https://github.com/TQRG/physalia

browser, and 20 repetitions for each trial, for a total running time of more than 18 hours.

Native experiments. The author of [2] used AR and its *mem-CPU* plugin for measuring the performance of 12 synthetic native apps we developed for assessing the run-time impact of 6 performance bugs supported by Android Lint. The experiment involves a laptop as base station, a phone running Android 8.0, and 20 repetitions for each trial. The duration of the experiment is about 12 hours.

AR as a learning platform. AR is used at different levels within the Computer Science program of the Vrije Universiteit Amsterdam. Firstly, AR is used for scaling up the projects within the Green Lab course, where teams of students design and conduct empirical studies on the energy efficiency of software; students use AR as a black-box tool for measuring real mobile apps without spending time on activities outside the learning objectives of the course (*e.g.,* writing and fixing bugs of boilerplate code). Students working on their final project go deeper on run-time profiling of Android apps by either developing new plugins or improving AR itself. This helps in (i) building a community of learners around AR and (ii) keeping AR always up to date and well tested. For partially mitigating accidental disruptive changes, a test suite is targeting all internal components of AR with 100% line code coverage.

5 BENCHMARKING AR

Table 3 shows the 27 benchmarking apps we developed for benchmarking AR. The first app is the baseline, it just loads an empty Android activity and it is generally used as baseline; then, each app is designed so to continuously perform the same operation until it is killed from AR. When it is possible to control the rate of the operation by the app (*i.e.,* Camera, CPU, GPS, Local storage, Networking, Room database), we developed three versions of the same app, each of them performing the operation at a specific interval.

Figure 3 shows one example of the results we obtained when benchmarking AR. In this case, we measured the energy consumption of all the 27 apps by using the *batterystats* plugin and two different Android devices, namely: a Xiaomi Mi 9T Pro (Android 10.0, 2.84GHz octo-core CPU, 6Gb of memory) and an LG Nexus 5X (Android 6.0, 1.8GHz heza-core CPU, 2Gb of memory). Every app and interval combination is run for 3 minutes and is then automatically stopped. Each run is repeated 30 times with 2 minutes between each run, in order to account for possible fluctuations of the measured energy consumption.

Even though in this paper we do not discuss the absolute precision of the measurement method behind the *batterystats* plugin (it has been already done in [3]), in Figure 3 we can see some interesting results. Firstly, the Nexus 5X tends to consume more energy with respect to the Mi 9T; we presume that this difference is mainly due to the fact that the Mi 9T is has hardware components of newer generation, like the display, CPU, network card, and sensors. We can also note that for the majority of the apps the coefficient of variation of the collected measures is very low (for 44 out of 54 apps it is below 5%); this is an indication of the reliability of the measurement infrastructure behind ARand its *batterystats* plugin. We can also note that some hardware components tend to consume more energy with respect to the baseline, they include the camera, display, and GPS (high frequency). Finally, we can also note that all networking apps exhibit a relatively higher degree of variation with respect to other apps; we presume that this variation is mostly due to the fact that even small fluctuations of the networking conditions can play a relatively important role in experiments involving mobile applications.

Table 3: Benchmarking Android apps

App	Description
Baseline	No functionality (idle)
Accelerometer	Listens for accelerometer data
Ambient light sensor	listens for changes in the ambient light sensor
Camera (x3)	Takes a picture from the front camera every 15, 10, or 5 seconds
CPU (x3)	Computes a large factorial number every 3, 2, or 1 seconds
Display	Plays a video at full brightness
GPS (x3)	Listens for location updates every 3, 2, or 1 seconds
Gravity sensor	Listens for continuous updates of the gravity sensor
Gyroscope	Listens for continuous updates of the gyroscope
Local storage (x3)	Writes 1 MB of data to local storage every 3, 2, or 1 seconds
Magnetic field sensor	Listens for continuous updates of the magnetic field sensor
Microphone	Records audio from microphone
Networking (x3)	Sends an HTTP Get requests every 3, 2, or 1 seconds
Room database (x3)	Writes 1 MB of data via the Room Android library every 3, 2, or 1 seconds
Speaker	Plays audio file at half volume

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented AR, a framework for streamlining the execution of measurement-based experiments targeting Android (web) apps. The added value of AR is to help researchers and practitioners in avoiding to reinvent the wheel with ad-hoc, not-replicable empirical pipelines. AR is customizable via external business logic and extensible thanks to a plugin-based architecture. In order to facilitate the validation of the accuracy of AR and similar frameworks, we also make available a set of 27 benchmarking Kotlin apps, each stressing specific hardware components of Android devices.

As future work, we are planning to (i) extend AR with additional hardware-based profilers, (ii) to further improve its customizability, (iii) to carry out more evaluations to determine the level of accuracy of the measurement results of AR, and (iv) to expand it towards other emerging software engineering subdomains like robotics.

REFERENCES

[1] Luciano baresi, William G. Griswold, Grace A. Lewis, Marco Autili, Ivano Malavolta, and Christine Julien. 2020. Trends and Challenges for Software Engineering in the Mobile Domain. *IEEE Software* (2020), to appear. https://doi.org/10.1109/MS.2020.2994306

[2] Teerath Das. 2020. *Investigating Performance Issues in Mobile Apps.* PhD Thesis. Computer Science Division, Gran Sasso Science Institute, L'Aquila, Italy.

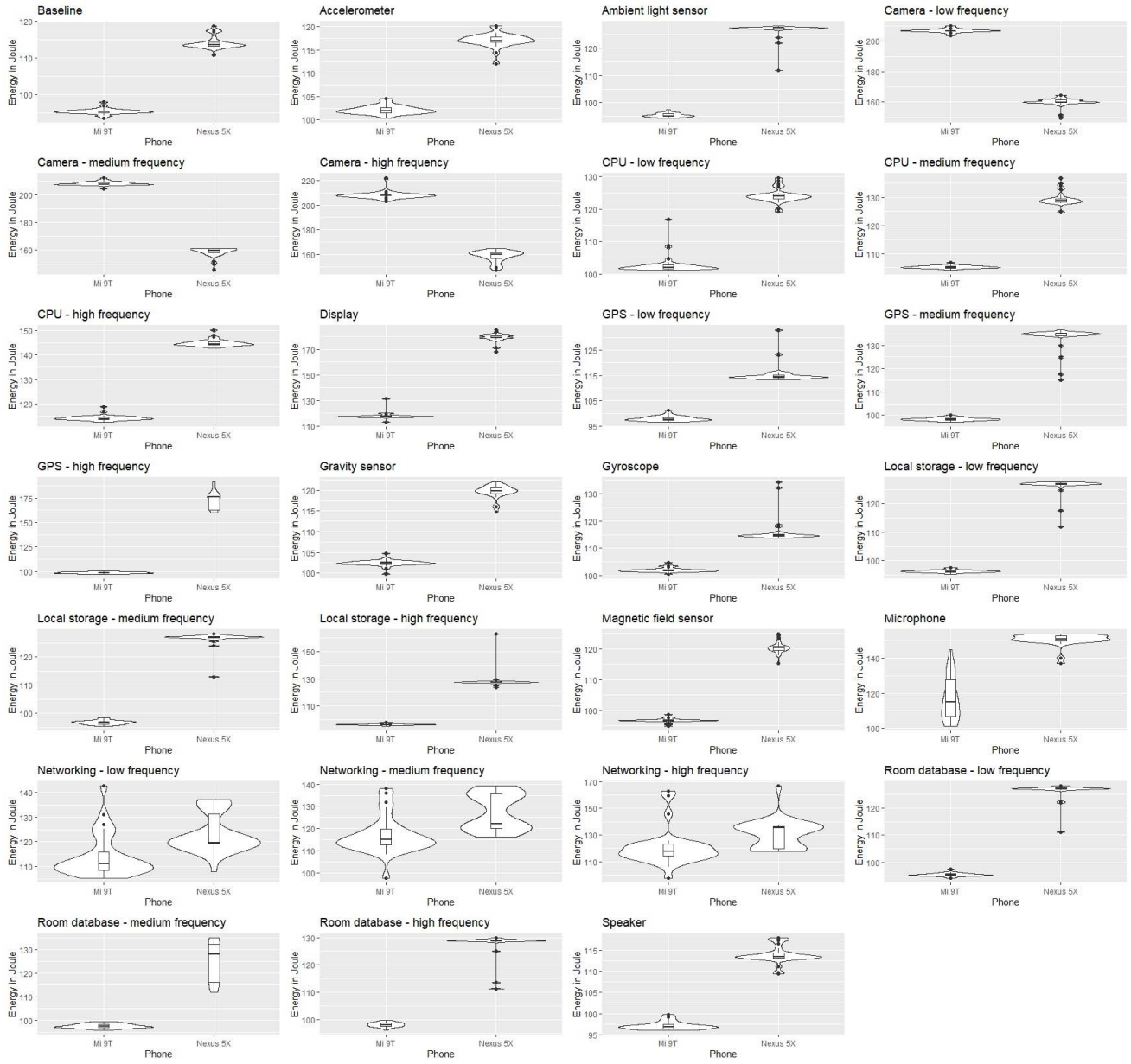

Figure 3: Benchmarking results of AR with the *batterystats* plugin

[3] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea 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)*. IEEE, 103–114.

[4] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An empirical study of the performance impacts of android code smells. In *Proceedings of the international conference on mobile software engineering and systems*. 59–69.

[5] Abram Hindle, Alex Wilson, Kent Rasmussen, E Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 12–21.

[6] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 399–410.

[7] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in android apps. In *international conference on software maintenance and evolution*. IEEE, 352–361.

[8] Ivano Malavolta, Katerina Chinnappan, Lukas Jasmontas, Sarthak Gupta, and Kaveh Ali Karam Soltany. 2020. Evaluating the Impact of Caching on the Energy Consumption and Performance of Progressive Web Apps. In *7th IEEE/ACM International Conference on Mobile Software Engineering and Systems 2020*.

[9] Javad Nejati and Aruna Balasubramanian. 2016. An in-depth study of mobile browser performance. In *International Conference on World Wide Web*. 1305–1315.

[10] Pijush Kanti Dutta Pramanik, Nilanjan Sinhababu, Bulbul Mukherjee, Sanjeevikumar Padmanaban, Aranyak Maity, Bijoy Kumar Upadhyaya, Jens Bo Holm-Nielsen, and Prasenjit Choudhury. 2019. Power Consumption Analysis, Measurement, Management, and Issues: A State-of-the-Art Review of Smartphone Battery and Energy Usage. *IEEE Access* 7 (2019), 182113–182172.