BatteryLab: A Collaborative Platform for Power Monitoring

https://batterylab.dev

Matteo Varvello[1(✉)], Kleomenis Katevas[2], Mihai Plesa[3], Hamed Haddadi[3], Fabian Bustamante[4], and Ben Livshits[5]

[1] Bell Labs Nokia, Holmdel, USA
matteo.varvello@nokia.com
[2] Telefonica Research, Madrid, Spain
kleomenis.katevas@telefonica.com
[3] Brave Software, Santa Clara, USA
{mplesa,hhaddadi}@brave.com
[4] Northwestern University, Evanston, USA
fabianb@cs.northwestern.edu
[5] Imperial College London, London, UK
b.livshits@imperial.ac.uk

Abstract. Advances in cloud computing have simplified the way that both software development and testing are performed. This is not true for battery testing for which state of the art test-beds simply consist of one phone attached to a power meter. These test-beds have limited resources, access, and are overall hard to maintain; for these reasons, they often sit idle with no experiment to run. In this paper, we propose to *share* existing battery testbeds and transform them into *vantage points* of BatteryLab, a power monitoring platform offering heterogeneous devices and testing conditions. We have achieved this vision with a combination of hardware and software which allow to augment existing battery test-beds with remote capabilities. BatteryLab currently counts three vantage points, one in Europe and two in the US, hosting three Android devices and one iPhone 7. We benchmark BatteryLab with respect to the accuracy of its battery readings, system performance, and platform heterogeneity. Next, we demonstrate how measurements can be run atop of BatteryLab by developing the "Web Power Monitor" (WPM), a tool which can measure website power consumption at scale. We released WPM and used it to report on the energy consumption of Alexa's top 1,000 websites across 3 locations and 4 devices (both Android and iOS).

Keywords: Battery · Test-bed · Performance · Android · iOS

1 Introduction

Power consumption is a growing concern in the mobile industry, ranging from mobile phone users, operating system vendors, and app developers. To accurately measure a device power consumption, two options are currently available:

O. Hohlfeld et al. (Eds.): PAM 2022, LNCS 13210, pp. 97–121, 2022.
https://doi.org/10.1007/978-3-030-98785-5_5

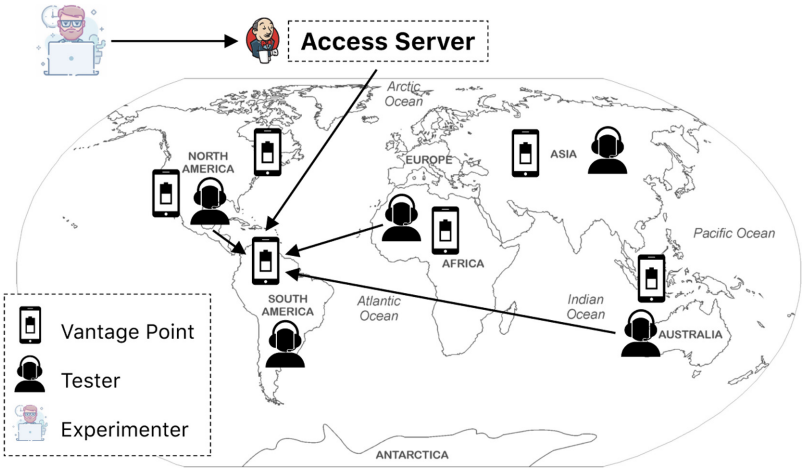

Fig. 1. Distributed architecture of BatteryLab.

software-based measurements, which rely on battery readings from the device, and *hardware-based* measurements which leverage an external power monitor connected to a device battery. Software-based power measurements are easy to use, but lack the accuracy and granularity an experimenter might require [15,38]. Few startups [20,27] offer, for a price, improvements upon the accuracy of software-based power measurements by relying on few devices for which they have performed heavy "calibration" (their secret sauce). Hardware-based power measurements are accurate, fine-grained, but quite cumbersome to setup.

For years, researchers have been building home-grown test-beds for hardware-based power measurements, consisting of an Android device connected to a high-frequency power monitor [13,14,21,40]. This required expertise in hardware setup and writing code when automation is needed – code which is unfortunately never shared with the community. Such closed-source test-beds have limited accessibility, e.g., requiring physical access to the devices, and shareability, even among members of the same group. This became clear during the COVID-19 pandemic: remote desktop tools like VNC came to the rescue, but often the only solution was to move that precious test-bed at home.

In this paper we challenge the assumption that such battery test-beds need to be "local" and propose *BatteryLab*, a cooperative platform for battery measurements. We envision BatteryLab as a cooperative platform where members contribute hardware resources (e.g., some phones and a power monitor) in exchange for access to the resources contributed by other platform members. Nevertheless, the hardware/software suite we have built and open sourced [25] can also be used "locally", i.e., augmenting an existing battery test-bed with scheduling and remote control capabilities. The following contributions are the founding blocks of BatteryLab:

Automation for Hardware-Based Power Measurements. BatteryLab comes with an intrinsic automation requirement. For example, an *experimenter* from Europe needs to be able to activate a power meter connected to a phone in the US. To make this possible, we have designed *vantage points* as the above local test-beds enhanced with a lightweight *controller* such as a Raspberry Pi [34]. The controller runs BatteryLab's software suite which realizes "remote power testing", e.g., from activating a device's battery bypass to enabling remote control of the device via the experimenter's browser.

A Library for Android and iOS Automation. While the Android Debugging Bridge (ADB) is a powerful tool to automate Android devices, an equivalent does not exist for iOS. BatteryLab builds atop of ADB to offer seamless automation of Android devices. For iOS, we have built and open-sourced a Python library which maps commands like touch, swipe, and text input to a (virtual) Bluetooth keyboard and mouse. To the best of our knowledge, we are the first to provide automation of any third party app on actual iOS devices (i.e., other than simulators as in [6,36]). Even commercial products for iOS, such as TeamViewer [39] or the recent SharePlay [7] of iOS 15, can only provide remote screen sharing.

Usability Testing for Power Measurements. BatteryLab allows an experimenter to interact with a real device via its browser. This feature is paramount for debugging automation scripts, but also a key enabler of *usability testing*, or battery measurements coupled with actual device interactions from real users.

Deployment at Three Research Institutions. BatteryLab currently has three vantage points, two in the US and one in Europe (with more vantage points going live soon) and hosts a range of Android devices and an iOS device (iPhone 7).

We evaluate BatteryLab on battery readings accuracy, system performance, and platform heterogeneity. To illustrate the value and ease-of-use of BatteryLab, we have also built the "Web Power Monitor" (WPM), a service which measures the power consumption of websites loaded via a test browser running at any BatteryLab's device. With a handful of lines of code, WPM allowed us to conduct the largest scale measurement study of energy consumption on the Web, encompassing Alexa's top 1,000 websites measured from four devices and two operating systems. We have released WPM as a web application integrated with BatteryLab which offers such testing capabilities to the public, in real time. This paper extends our previously published work [44] in many ways:

- We add support for device automation also to Apple iOS by exploiting the Bluetooth HID and AirPlay services.
- We deploy BatteryLab at three research institutions and benchmark its performance including, among others, a comparison with software-based battery measurements.
- We open source BatteryLab's code for "local" use, and BatteryLab as a testbed for battery measurements.

– We develop and release WPM, a tool for measuring website power consumption at scale; we further use WPM to measure the energy consumption of Alexa's top 1,000 websites across 3 locations and 4 devices.
– We explore support for usability testing via "action replay", a mechanism to automatically build app automation scripts based on human inputs.

2 BATTERYLAB Architecture

This section presents the design and implementation details of BatteryLab (see Fig. 1). Our current iteration focuses on mobile devices, but the architecture is flexible and can be extended to other devices, e.g., laptops and IoT devices.

BatteryLab consists of a centralized *access server* that remotely manages a number of nodes or *vantage points*. Each of these vantage points, hosted by universities or research organizations around the world, includes a number of test devices (a phone/tablet connected to a power monitor) where experiments are carried out. BatteryLab members (*experimenters*) gain access to test devices via the access server, where they can request time slots to deploy automated scripts and/or remote control of the device. Once granted, remote device control can be shared with *testers*, whose task is to manually interact with a device, e.g., scroll and search for items on a shopping application. Testers are either volunteers, e.g., recruited via email or social media, or paid, recruited via crowdsourcing websites like Mechanical Turk [3].

In the remainder of this section, we describe BatteryLab's main components in detail. Next, we focus on BatteryLab's automation capabilities and on the procedure for new members to join the platform.

2.1 Access Server

The main role of the access server is to manage the vantage points and schedule experiments on them based on experimenters' requests. We built the access server atop of the Jenkins [22] continuous integration system which is free, open-source, portable (written in Java) and backed by an active and large community. Jenkins enables end-to-end test pipelines while supporting multiple users and concurrent timed sessions.

BatteryLab's access server runs in the cloud (Amazon AWS) which enables further scaling and cost optimization. Vantage points have to be added explicitly and pre-approved in multiple ways (IP lockdown, security groups). Experimenters need to authenticate and be authorized to access the web console of the access server, which is only available over HTTPS. The access server communicates with the vantage points via SSH. New BatteryLab members grant SSH access from the server to the vantage point's controller via public key and IP white-listing (Sect. 3.4).

Experimenters access vantage points via the access server, where they can create *jobs* to automate their tests. Jobs are programmed using a combination of BatteryLab's Python API (Table 1), e.g., for user-friendly device selection

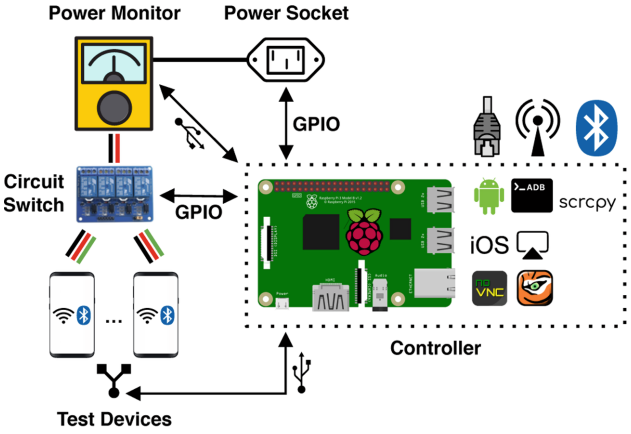

Fig. 2. Vantage point design.

and interaction with the power meter, and code specific to each test. Only the experimenters who have been granted access to the platform can create, edit, or run jobs and every pipeline change has to be approved by an administrator. This is done via a role-based authorization matrix.

After the initial setup, the access server dispatches queued jobs based on the experimenter constraints, e.g., target device, connectivity, or network location, and BatteryLab constraints. For example, no concurrent jobs are allowed at the same vantage point since the power monitor can only be associated with one device at a time and isolation is required for accurate power measurements. By default, the access server collects logs from the power meter which are made available for several days within the job's workspace. Android logs (e.g., `logcat` and `dumpsys`) can be requested via the `execute_command` API for the supported devices (Table 1).

2.2 Vantage Point

Figure 2 shows a graphical overview of a BatteryLab's vantage point with its main components: controller, power monitor, test devices, circuit switch, and power socket.

Controller – This is a Linux-based machine responsible for managing the vantage point. This machine is equipped with both Ethernet, WiFi and Bluetooth connectivity, a USB controller with a series of available USB ports, as well as with an external General-Purpose Input/Output (GPIO) interface. We use the popular Raspberry Pi 3B+ [34] running Raspberry Pi OS (Buster, September 2019) that meets these requirements at an affordable price.

The controller's primary role is to manage connectivity with test devices. Each device connects to the controller's USB port, WiFi access point (configured in NAT or Bridge mode), and Bluetooth, based on automation needs (see

Sect. 3.2). USB is used to power each testing device when not connected to the power monitor and to instrument Android devices via the Android Debugging Bridge [19] (ADB), when needed. WiFi provides Internet access to all devices and extend ADB automation and device mirroring to Android devices without incurring the extra USB current, which interferes with the power monitoring procedure. (De)activation of USB ports is realized using `uhubctl` [43]. Bluetooth is used to realize automation across OSes (Android and iOS) and connectivity (WiFi and cellular).

The second role of the controller is to provide *device mirroring*, i.e., remote control of device under test. We use VNC (`tigervnc` [41]) to enable remote access to the controller, and `noVNC` [31], an HTML VNC library and application, to provide easy access to a VNC session via a browser without additional software required at the experimenter/tester. We then *mirror* the test device within the noVNC/VNC session and limit access to only this visual element. In Android, this is achieved using `scrcpy` [18], a screen mirroring utility which runs atop of ADB for devices running API 21 (Android $\geq$ 5.0). In iOS, we utilize AirPlay Screen Mirroring [8] using `RPiPlay` [17], an AirPlay mirroring server for devices running iOS $\geq$ 9.0.

We have also built a graphical user interface (GUI) around the default `noVNC` client. The GUI consists of an *interactive area* where a device screen is mirrored (bottom of the figure) while a user (experimenter or tester) can remotely mouse-control the physical device, and a *toolbar* that occupies the top part of the GUI and implements a convenient subset of BatteryLab's API (see Table 1).

Power Monitor – This is a power metering hardware capable of measuring the current consumed by a test device in high sampling rate. BatteryLab currently supports the Monsoon HV [29], a power monitor with a voltage range of 0.8V to 13.5V and up to 6A continuous current sampled at 5 KHz. The Monsoon HV is controlled using its Python API [30]. Other power monitors can be supported, granted that they offer APIs to be integrated with BatteryLab's software suite.

Test Device(s) – It is an Android or iOS device (phone or tablet) that can be connected to a power monitor using a battery bypass modification (i.e., isolate the battery power circuit and provide power via the power monitor). While devices with removable batteries are easier to setup, more complex configurations (e.g., all iOS and recent Android devices) are also supported by doing the battery bypass modification at the battery controller level.

Circuit Switch – This is a relay-based circuit with multiple channels that lies between the test devices and the power monitor. The circuit switch is connected to the controller's GPIO interface and all relays can be controlled via software from the controller. Each relay uses the device's voltage (+) terminal as an input, and programmatically switches between the battery's voltage terminal and the power monitor's `Vout` connector. Ground (-) connector is permanently connected to all devices' Ground terminals.

This circuit switch has three main tasks. First, it allows to switch between a direct connection between the phone and its battery, and the "battery bypass"—

Table 1. BatteryLab's core API.

API	Description	Parameters
list_nodes	List matching vantage points	label, state
list_devices	List identifiers of test devices	vantage_point
device_mirroring	Activate device mirroring	device_id
power_monitor	Toggle Monsoon power state	state (on/off)
set_voltage	Set target voltage	voltage_val
start_monitor	Start battery measurement	device_id, duration
stop_monitor	Stop battery measurement	-
batt_switch	(De)activate battery	device_id
execute_command	Execute a command on device	device_id, command, automation

which implies disconnecting the battery and connecting to the power monitor. This is required to allow the power monitor to measure the current consumed during an experiment. Second, it allows BatteryLab to concurrently support multiple test devices without having to manually move cables around. Third, it allows to programmatically switch the power meter on and off.

Power Socket – This is a relay-based power socket that allows the controller to turn the Monsoon on and off, when needed. It connects to the controller via the GPIO port, and it is controlled by our Python API.

3 Using BatteryLab

In the following paragraphs we illustrate the use of BatteryLab's API, discuss its support of test automation, and the generation of automation scripts from human input. We close the section with a description of the steps needed to join BatteryLab.

3.1 API Usage

Experimenter jobs are interleaved with "control" jobs which manage the vantage points, e.g., they update BatteryLab wildcard certificates (Sect. 3.4) and ensure that the power meter is not active when not needed (for safety reasons). We here present some of these jobs as examples of BatteryLab's API usage. We have chosen the set of jobs that are also used by the application we have built

atop of BatteryLab (Sect. 5). Note that these jobs effectively *extend* the API available to BatteryLab's experimenters; these are not listed in Table 1 which focuses only on core API.

`NODE_SETUP` – The goal of this job/API is to prepare a vantage point for power measurements on a device d. This implies activating the power meter (`power_monitor`), offering the voltage that d requires (`set_voltage`) and activating the relay to realize d's battery bypass (`batt_switch`). The job continues by verifying that WiFi is properly working, eventually switching frequency based on the device characteristics—with 5 GHz preferred, when available. Based on the device and the requested automation (see Sect. 3.2), the job continues by either activating ADB over WiFi or the Bluetooth HID service. Finally, USB connection is interrupted—to avoid noise on the power measurements—and device mirroring is activated, if needed (`device_mirroring`).

`DEVICE_SETUP` – The goal of this job/API is to prepare a device d such that "noise" on the upcoming power measurement is minimized. We have identified several best practices which help in doing so and we offer them as an API. Nevertheless, the experimenter is the ultimate decision maker and can either ignore or further improve on these operations. The job starts by disabling notifications, set the device in airplane mode with WiFi only activated—unless a mobile connection is needed and available—and close all background apps. Next, the job ensures that the device is not using automatic brightness and further sets the brightness to a default value or a requested one. The last step is important since the variation in ambient light can impact the outcome of a measurement.

`CLEANUP` – The goal of this job/API is to ensure that a vantage point is in a "safe" state. This implies turning off the power meter if no testing job is undergoing and removing any eventual battery bypass. Finally, USB connectivity is re-enabled which ensures that the device's battery get charged. This job further proceeds removing installed apps which were not used in the last seven days, with the goal to avoid overloading testing devices.

`REFRESH` – The goal of this job/API is to verify reachability of vantage points and devices therein. The information collected is used to populate a JSON file which enhances Jenkins data past sites reachability via SSH. This job currently runs across the whole platform every 30 min.

3.2 Android/iOS Automation Library

BatteryLab provides a Python library—which we open-sourced together with the BatteryLab's code—that greatly simplifies test automation on both Android and iOS. At high level, the library offers APIs like `input(tap, x, y)` which map to several underlying automation mechanisms, each with its own set of advantages and limitations. The library automatically switches to an automation solution based on the experiment needs, e.g., device and connectivity, hiding unnecessary complexity to the experimenter.

Android Debugging Protocol (Android) – ADB [19] is a powerful tool/protocol to control an Android device. Commands can be sent over USB, WiFi, or Bluetooth. While USB guarantees highest reliability, it interferes with the power monitor due to the power required to activate the USB micro-controller at the device. Accordingly, BatteryLab's automation library uses ADB over USB *whenever* the power monitor is not used, e.g., when installing an app or cleaning a device, while resorting to WiFi (or Bluetooth) for all other automations. Note that using WiFi implies not being able to run experiments leveraging the mobile network. However, these experiments are possible leveraging Bluetooth tethering, when available.

Bluetooth HID Service (iOS/Android) – Automating third-party apps in iOS is challenging due to the lack of ADB-like API. Even commercial solutions like TeamViewer [39] or the new SharePlay [7] of iOS 15 limit their iOS offering to remote screen viewing only. The only solution to control an iOS device without physical access requires using a wireless keyboard and mouse. We exploit this feature to map commands like touch, swipe and text input into (virtual) mouse and keyboard actions.

Specifically, we virtualize the mouse and keyboard by designing a Human Interface Device (HID) service [11] atop of BlueZ Bluetooth Protocol Stack [12] v5.43. The controller broadcasts a custom Combo Keyboard/Pointing HID service (i.e., HIDDeviceSubclass: `0xC0` [11]) which enables a connection to previously paired test devices over Bluetooth. The automation library translates keyboard keystrokes, mouse clicks and gestures into USB HID Usage Reports [42] that simulate user actions to the controlled device (e.g., locate an app, launch it, and interact with it). While we exploit this automation strategy for iOS only, the approach is generic and can be used across all devices which support the Bluetooth HID profile for both mouse and keyboard (i.e., Android v8.0+ and iOS v13.0+).

3.3 Action Replay

Regardless of the automation mechanism used, building automation scripts for mobile devices is a time consuming task [24,32,35]. Device mirroring offers a unique opportunity to speed up the generation of such automation scripts in BatteryLab. The key idea is to record an experimenter/tester clicks, mouse, keyboard input, and use them to generate an automation script.

We have thus modified noVNC – precisely `mouse.js` and `keyboard.js` – to POST the collected user input to the controller's web application (see Sect. 2.2) where the device being mirrored is hosted. The web application collects the user input and map it to APIs from the above automation library, which translates into, for example, an ADB command such as `tap` or `swipe`. When screen coordinates are involved, e.g., for a `tap` command, the actual coordinates are derived by offsetting the coordinates recorded in noVNC as a function of the size of the VNC screen and the actual device size. Under the assumption that an application GUI is similar across platforms, the human-generated automation script at a given device could be re-used for other devices.

3.4 How to Join?

Joining BatteryLab is straightforward and consists of three steps. First, the vantage point needs to be physically built as described in Fig. 2. At this point, the controller (Raspberry Pi) should also be flashed with the latest Raspberry Pi OS image along with some standard setup as described in the associated tutorial [9]. Second, the network where the controller is connected (via Ethernet) needs to be configured to allow the controller to be reachable at the following configurable ports: 2222 (SSH, access server only)[1], 8080 (web application for GUI and action/replay). Third, a BatteryLab account should be created for the new member. This involves downloading the access server's public key—to be authorized at the controller—and uploading a human readable identifier for the vantage point (e.g., `node1`), and its current public IP address. This information is used by the access server to add a new entry in BatteryLab's DNS (e.g., `node1.batterylab.dev`)—provided by Amazon Route53 [5]—and verify that SSH access to the new vantage point is now granted. Since the whole BatteryLab traffic is encrypted, a wildcard `letsencrypt` [23] certificate is distributed to new members by the access server, which also manages its renewal and distribution, when needed.

The next step consists of installing BatteryLab's software at the controller. This step is realized automatically by BatteryLab's access server and it is the first job to be deployed at the new vantage point. At high level, this consists in the following operations. First, the OS is updated. Next, common security practices are enforced: 1) install `fail2ban` which neutralizes popular brute-force attacks over SSH, and 2) disable password authentication for SSH. Next, BatteryLab code is pulled from its open source repository [25] along with all packages and software needed. Code is compiled, where needed, and packages are installed. Then, the controller is turned into an "access point" where the test devices will connect to. By default, the access point spins a new SSID (`BatteryLab`) with a pre-set password operating on 2.4 GHz. However, BatteryLab automatically switches to 5 GHz for devices that support it. This "switch" is required since the Raspberry Pi does not mount two WiFi antennas and thus both frequencies cannot be active at the same time.

Next, several `crontab` entries are added. At reboot and every 30 min, a task monitors the controller's public IP address and update its entry at BatteryLab's DNS. At reboot, the GPIO pins used by BatteryLab are set as "output" and the IP rules needed by the controller to act as an access point are restored. The next step consists in setting up device mirroring, i.e., VNC password and wildcard certificate used by both noVNC and the web application. This setup job also learns useful information about the devices connected: ADB identifier (if available), screen resolution, IP address, etc. This information is reported to the access server to further populate the JSON file maintained by the `REFRESH` job/API. Last but not least, several tests are run to verify: 1) Monsoon connectivity, 2) device connectivity, 3) circuit relay stability, 4) device mirroring.

[1] The SSH agent at the node also needs to be configured accordingly. An iptable rule should be added to limit access to the access server only.

Table 2. BatteryLab test-bed composition.

	J7DUO	IPHONE7	SMJ337A	LMX210
Vendor	Samsung	Apple	Samsung	LG
OS	Android 9.0	iOS 13.2.3	Android 8.0.0	Android 7.1.2
Location	United Kingdom	United Kingdom	New Jersey	Illinois
CPU Info	Octa-core (2x2.2 GHz Cortex-A73, 6x1.6 GHz	Quad-core 2.34 GHz Apple A10 Fusion	Quad-core 1.4 GHz Cortex-A53	Quad-core 1.4 GHz Cortex-A53
Memory	4 GB	2 GB	2 GB	2 GB
Battery	3,000 mAh	1,960 mAh	2,600 mAh	2,500 mAh

BatteryLab currently counts three vantage points located in the UK, New Jersey, and Illinois, with a total of three Android devices and one iPhone 7. Table 2 provides detailed information of the devices currently available to the public via BatteryLab. At the time of writing, three other organizations are in the process of setting up a BatteryLab vantage point.

4 Benchmarking

This section benchmarks BatteryLab. We first evaluate its *accuracy* in reporting battery measurements. We then evaluate its *performance* with respect to CPU, memory, and responsiveness of its device mirroring mechanism. We then investigate BatteryLab's *heterogeneity* and the feasibility of usability testing when coupled with power monitoring.

4.1 Accuracy

Compared to a classic *local* setup for device performance measurements, BatteryLab introduces some hardware (circuit relay) and software (device mirroring) components that can impact the *accuracy* of the measurements. We devised an experiment where we compare three scenarios. First, a *direct* scenario consisting of the Monsoon power meter, the testing device, and the Raspberry Pi to instrument the power meter. For this setup, we strictly followed Monsoon indications [29] in terms of cable type and length, and connectors to be used. Next we evaluate a *relay* scenario, where the relay circuit is introduced to enable BatteryLab's programmable switching between battery bypass and regular battery operation (see Sect. 2.2). Finally, a *mirroring* scenario where the device screen is mirrored to an open noVNC session. While the relay is always "required" for BatteryLab to properly function, device mirroring is only required for usability testing. Since we currently do not fully support usability testing for iOS (see Sect. 2.1), we here only focus on Android.

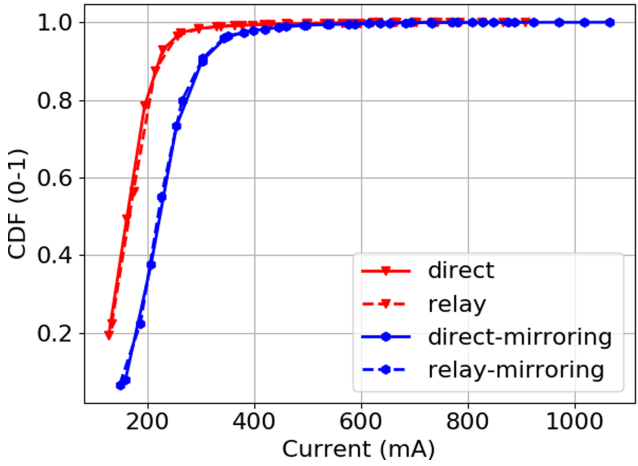

Fig. 3. CDF of current drawn (direct, relay, direct-mirroring, relay-mirroring).

Figure 3 shows the Cumulative Distribution Function (CDF) of the current consumed in each of the above scenarios during a 5 min test. For completeness, we also consider a *direct-mirroring* scenario where the device is directly connected to Monsoon and device mirroring is active. During the test, we play an MPEG4 video pre-loaded on the SD card of the device (J7DUO, UK). The rationale is to force the device mirroring mechanism to constantly update as new frames are initiated. The figure shows negligible difference between the "direct" and "relay" scenarios, regardless of the device mirroring status being active or not. A larger gap (median current grows from 160 to 220 mA) appears with device mirroring. This is because of the background process responsible for screencasting to the controller which causes additional CPU usage on the device (∼15%). At the end of this section, we investigate a more challenging usability testing scenario along with a potential solution to minimize the additional power consumption caused by device mirroring.

A related question is: *what is the accuracy that BatteryLab offers compared to software measurements?* Having verified that BatteryLab is as accurate as a local setup, and granted that hardware-based battery measurements are the "ground truth", the question is really how accurate are software-based battery measurements? While this question is out of scope for this paper, it has to be noted that Android software-based battery readings can be realized in BatteryLab via ADB[2]. With respect to iOS, while some high level *energy usage* reports are available—reporting battery consumption every second on an arbitrary 0 to 20 scale—they are currently unavailable to BatteryLab since they require a developer-enabled macOS.

[2] Either using Android bug-report files or with `adb shell cat sys/class/power_supply/*/uevent`.

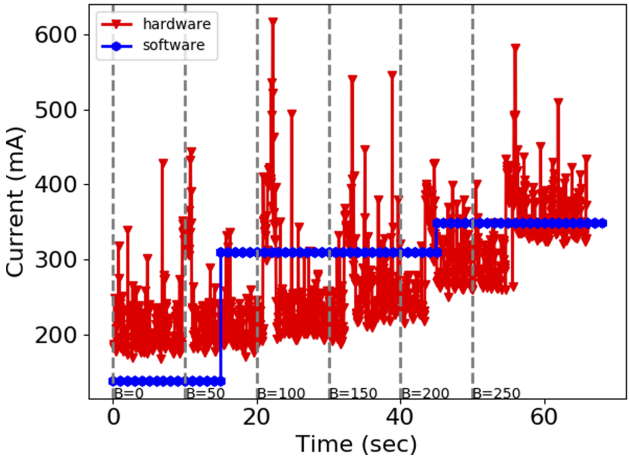

Fig. 4. Current over time under variable screen brightness (B) from 0 to 250, Android's max value. Software versus hardware measurements (LMX210).

We find that pure software measurements are *enough* to identify trends in measured current, but have limited overall accuracy and granularity, e.g., a 30 s reporting frequency across all our Android devices. As an example, Fig. 4 shows the time evolution of the current measured via BatteryLab (hardware) and software while increasing the screen brightness from minimum (0) to maximum (250) by 50 units over 60 s, as indicated by the vertical dashed lines. This plot shows that pure software measurements are *enough* to identify trends, but have limited overall accuracy and granularity – while the plot refers to the LMX210, we measured a similar reporting frequency (30 s) across all Android devices.

To further investigate the reporting frequency, we have performed the same test also on Samsung's Remote Test Lab [37].[3] We find a 10 s reporting frequency on Samsung Galaxy S5 (Android 6) and S7 (Android 8), and 30 s on S8 and S9 (Android 9). When repeating the same tests on newer models, we find that the reported sampling rate improves to a mean of 2.23 s (±1.65) for Google Pixel 3a (Android 12), 0.66 (±0.24) for Google Pixel 4 (Android 12) and 0.60 (±0.25) for Google Pixel 5. The sampling rate was unaffected from different configurations (screen on, off, or streaming a HD video). Note that internal battery readings can be enhanced with additional data (e.g., cpu, screen usage), alongside device calibration, to achieve higher accuracy, as discussed in [15].

4.2 System Performance

Next, we benchmark overall BatteryLab performance. We start by evaluating the CPU utilization at the controller. Figure 5 shows the CDF of the CPU utilization during the previous experiments (when a relay was used) with active and inactive device mirroring, respectively. When device mirroring is inactive, the controller is

[3] These tests were not possible on AWS Device Farm [4] due to lack of ADB access.

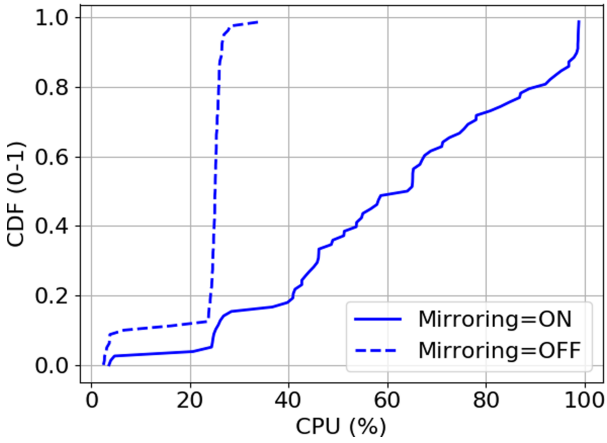

Fig. 5. CDF of CPU consumption at the controller (Raspberry Pi 3B+)

mostly underloaded, i.e., constant CPU utilization at 25%. This load is caused by the communication with the power meter to pull battery readings at the highest frequency (5 kHz). With device mirroring, the median load increases to ∼75%. Further, in 10% of the measurements the load is quite high and over 95%.

Device mirroring only impacts the CPU usage. The impact on memory consumption is minimal (extra 6%, on average). Overall, memory does not appear to be an issue given less than 20% utilization of the Raspberry Pi's 1 GB. The networking demand is also minimal, with just 32 MB of upload traffic for a ∼7 min test (due to device mirroring). Note that we set `scrcpy`'s video encoding (H.264) rate to 1 Mbps, which produces an upper bound of about 50 MB. The lower value depends on extra compression provided by `noVNC`.

Finally, we investigate the "responsiveness" of device mirroring. We call *latency* the time between when an action is requested (either via automation or a click in the browser), and when the consequence of this action is displayed back in the browser, after being executed on the device. This depends on a number of factors like network latency (between the browser and the test device), the load on the device and/or the controller, and software optimizations. We estimate such latency by recording audio (44,100 Hz) and video (60 fps) while interacting with the device via the browser. We then manually annotated the video using ELAN multimedia annotator software [46] and compute the latency as the time between a mouse click (identified via sound) and the first frame with a visual change in the app. We repeat this test 40 times while co-located with the vantage point (1 ms network latency) and measure an average latency of 350 (±80) ms.

4.3 Devices and Locations

BatteryLab's distributed nature is both a *feature* and a *necessity*. It is a feature since it allows battery measurements under diverse device and network

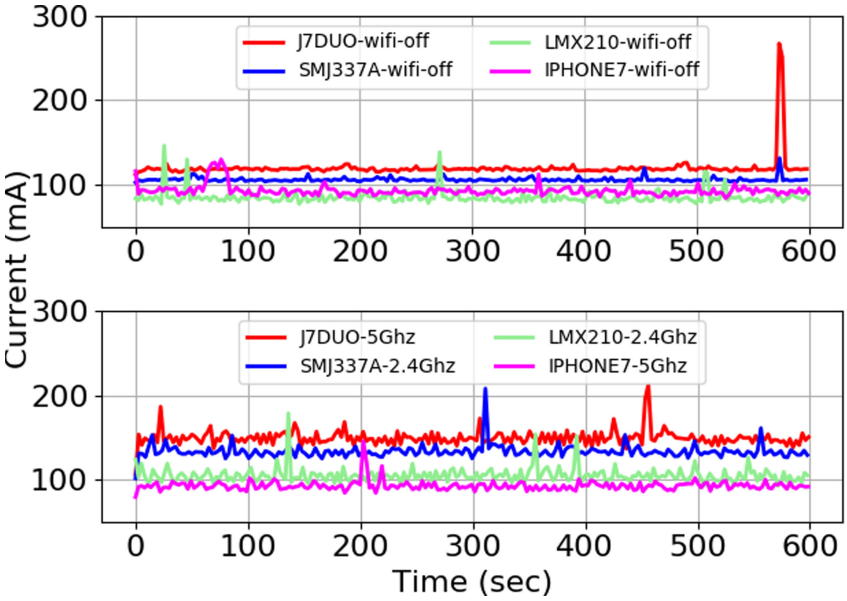

Fig. 6. Time evolution of current usage per device at rest (WiFi off and on).

conditions which is, to the best of our knowledge, a first for research and development in this space. It is a necessity since it is the way in which the platform can scale without incurring high costs. We here explore the impact of such diversity on battery measurements.

Figure 6 displays the evolution over time (600 s) of the current used by each BatteryLab device at "rest", i.e., displaying the default phone desktop after having run BatteryLab's API `DEVICE_SETUP` (Sect. 2.1) to ensure equivalent device settings. We further differentiate between the case when WiFi was active or not. For Android, regardless of WiFi settings, the figure shows that the J7DUO consumes the most, while the LMX210 consumes the least – about 25% less (270 vs 359 J over 600 s). Overall, the similar results in the case without WiFi suggest that the difference between the device is intrinsic of the device configurations, e.g., more power-hungry hardware and different Android versions with potential vendor customization. Understanding the event responsible of the variations shown in Fig. 6 is out of the scope of this analysis. It is worth noticing some correlations between peaks suggesting vendor specific operations, e.g., the peak around 580 s for the J7DUO and SMJ337A, both Samsung devices. The IPHONE7 consumes the least when considering active WiFi, while the LMX210 consumes the least in absence of WiFi. The take away of this analysis is that BatteryLab's devices (and locations) have the potential to offer a large set of heterogeneous conditions for the experimenters to test with.

Next, we compare the performance of the *same* device at different locations. Since we do not have such testing condition, we emulate the presence of one

Table 3. ProtonVPN statistics.

Speedtest server (Kms)	Download (Mbps)	Upload (Mbps)	Latency (ms)
South Africa Johannesburg (3.21)	6.26	9.77	222.04
China Hong Kong (4.86)	7.64	7.77	286.32
Japan Bunkyo (2.21)	9.68	7.76	239.38
Brazil Sao Paulo (8.84)	9.75	8.82	235.05
CA, USA Santa Clara (7.99)	10.63	14.87	215.16

device (J7DUO) at different locations via a VPN. We use a basic subscription to ProtonVPN [33] set up at the controller. Table 3 summarizes five locations we choose, along with network measurements from SpeedTest (upload and download bandwidth, latency). VPN vantage points are sorted by download bandwidth, with the South Africa node being the slowest and the California node being the fastest. Since the SpeedTest server is always within 10 km from each VPN node, the latency here reported is mostly representative of the network path between the vantage point and the VPN node.

Next, we leverage WPM (see Sect. 5) to investigate the battery consumption of Chrome in comparison to a new privacy-preserving browser (Brave). We assume a simple workload where each browser is instrumented to sequentially load 10 popular news websites. After a URL is entered, the automation script waits 6 s – emulating a typical page load time (PLT) – and then interact with the page by executing multiple "scroll up" and "scroll down" operations.

Figure 7 shows the average energy consumption (J) over 5 runs (standard deviation as errorbars) per VPN location and browser. The figure does not show significant differences among the battery measurements at different network location. For example, while the available bandwidth almost doubles between South Africa and California, the average discharge variation stays between standard deviation bounds. This is encouraging for experiments where BatteryLab's distributed nature is a *necessity* and its noise should be minimized.

Figure 7 also shows an interesting trend when comparing Brave and Chrome when tested via the Japanese VPN node. In this case, Brave's energy consumption is in line with the other nodes, while Chrome's is minimized. This is due to a significant (20%) drop in bandwidth usage by Chrome, due to a systematic reduction in the overall size of ads shown in Japan. This is an interesting result for experiments where BatteryLab's distributed nature is a *feature*.

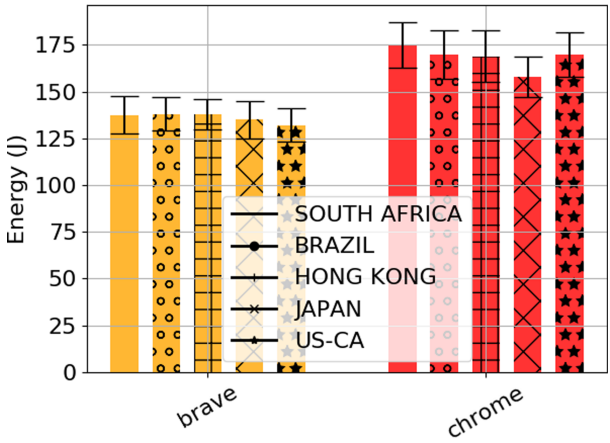

Fig. 7. Brave and Chrome energy consumption measured through VPN tunnels.

4.4 Usability Testing

The above analysis indicates that the extra (CPU) cost of device mirroring can invalidate the power measurements reading. Accordingly, our recommendation is to only leverage device mirroring when debugging an application over BatteryLab, but then disable it during the actual power measurements. This is not possible in case of usability testing where, by definition, a remote tester requires access to a device.

"Action replay" (see Sect. 3.3) is a potential solution to this limitation. The intuition is that a usability test can be split in two parts. First, the tester performs the required test while action replay is used to record her actions. Then, the tester's actions are replayed without the extra cost associated with device mirroring. While this approach provides, in theory, better accuracy, record and replay of human actions is challenging.

Figure 8 shows current measured over time at SMJ337A when a human interacts with the news workload in Brave (*action* curve), versus a bot generated by the action replay tool (*replay* curve). The figure shows overall lower current usage in the replay case versus the action case, resulting in an overall lower energy consumption: 345 versus 399 J over 380 s. The figure also shows high correlation between the two curves, suggesting accurate replay. Nevertheless, the overall error for energy estimation is fairly low (about 10%) and overall constant, similar to what suggested by Fig. 3 and now confirmed in a more challenging scenario.

Record and replay of human actions is generally challenging. Not all applications behave equally between successive runs. For example, in our testing scenario, a webpage could suddenly slow down and the replay might fail to click on an article that was not loaded yet. This opens up a potential interesting research area of building ML-driven tools for app testing based on human input,

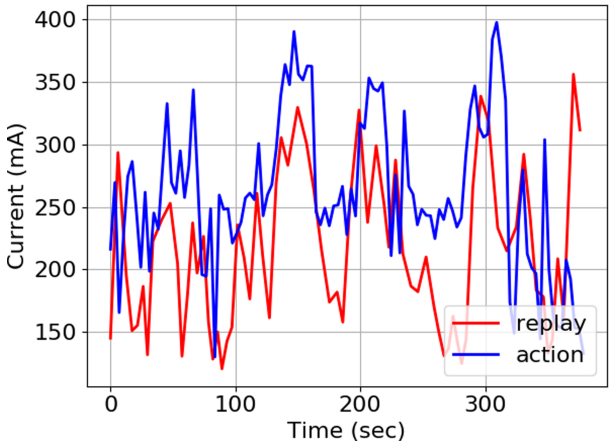

Fig. 8. Evolution over time of current (mA) measured during "action" (one human interacting with 10 news websites in sequence) and "replay" (a bot reproducing human activity).

as previously discussed in [2]. This is currently out of the scope of this work, but pairing such research with device behavior monitoring is an interesting avenue for future work.

5 The Web Power Monitor

There is an increasing interest in understanding the power drawn by modern websites and browsers [13,14,40], especially on smartphones due to inherent battery constraints. Although different, these studies share a *scalability* limitation: 1) they most target a single Android device, 2) they only test 100 websites or less accessed from a single location. This is because of the intrinsic limitations of the test-bed used which BatteryLab aims at solving. In the following, we present the "Web Power Monitor" (WPM), a BatteryLab application which enable large scale measurements of energy consumption in the Web. WPM currently powers a Web service [10] which offers such testing capabilities to the public, in real time.

5.1 Design and Implementation

Back-end – This is a Jenkins job whose goal is to report on the power consumed by a webpage when loaded on a BatteryLab device via a desired browser. Algorithm 1 shows the pseudocode describing such job; for simplicity, we omit the part of the job that takes care to identify *where* this test should run, i.e., either a specific device or a vantage point. As a first step (L1), the job prepares the vantage point by calling `NODE_SETUP` (see Sect. 3.1) which activates power monitoring and device mirroring, if requested. Next, the device is prepared for the test using `DEVICE_SETUP` (L2), also described in Sect. 3.1.

Algorithm 1: Pseudocode for WPM's backend.

Input: Device *device*, URLs to be tested *url_list*, Browser *browser*, Number of repetitions *reps*, Power flag *power*, Visual flag *visual*, Automation *automation*

Output: JSON file with performance metrics

```
1 node_status ← NODE_SETUP(power, visual)
2 device_status ← DEVICE_SETUP(device)
3 for r ← 0 to reps do
4 |   BROWSER_SETUP(device, browser)
5 |   RUN_TEST(device, browser, url_list, automation)
6 end
7 device_status ← CLEANUP(device)
```

Next is `BROWSER_SETUP` (L3), where the browser is i) installed (if needed), ii) cleaned (cache and configuration files), and iii) freshly started, e.g., Chrome requires to go through an onboarding process when launched for the first time. This function is equivalent to what an experimenter would have to design for a local experiment, with the caveat that it has to be tested on a range of devices. BatteryLab further simplifies this task for an experimenter via the "action replay" module (see Sect. 3.3) which generates automation scripts from human input collected via BatteryLab's browser interface.

The next step is `RUN_TEST` (L5) where a list of URLs (*url_list*) is tested. The experimenter defines how URLs should be loaded, e.g., sequentially in a new tab. The experimenter also controls whether to perform a *simple load*, i.e., load the page for a fixed amount of time, or interact with each page, e.g., scroll the page up and down multiple times for a certain duration. Finally, `CLEANUP` (see Sect. 3.1) is invoked to restore the node state (e.g., turn off the power meter) and, if needed, expose the collected data to the front-end.

Front-end – This is inspired by `webpagetest` [45], a tool for measuring webpage load times. Similarly, WPM offers a simple Web interface where a visitor can choose the URL to test, the browser, the device (along with *where* this device is located in the world) and which test to run. Power measurements are always collected, while visual access to the device needs to be explicitly requested. If so, the front-end alerts the user that this condition might impact the absolute value of the measurements, as discussed in Sect. 4.

Once the requested test is submitted, the user is presented with a page showing the progress of the experiment. If visual access was requested, as `DEVICE_SETUP` is completed, an iframe is activated on the page to show the requested device in real time. At the end of `RUN_TEST`, several plots are shown on screen such as CPU and current consumption during the test. We invite the interested reader to try out WPM at [10].

Front-end information (locations, devices, and browsers available) is maintained by the `REFRESH` job (see Sect. 3.1) and then retrieved as a JSON file via an

Table 4. WPM's measurement study versus previous works.

Study	OSes	Devices	Websites	Locations
[40]	Android	1	25	1
[14]	Android	1	80	1
[13]	Android	1*	100	1
WPM	Android/iOS	4	715	3

AJAX call. Similarly, the status of the experiment is pulled over time to switch between showing the remote device or results, when available.

5.2 Results

We used WPM to study the power consumption of Alexa's top 1,000 websites from 3 Android devices (NJ, IL, and UK) and one iPhone 7 (UK). We assume a simple load, where each page is loaded for up to 30 s. We load each page 3 times (with browser cache cleanup in between), and then report on the median for each metric. We synchronize experiments at the different locations using a 2 min fixed duration. The iOS test was not synchronized (one day delay) since only one testing device can be measured at a time per location.

To limit the scope (and duration) of this test, we only experiment with a single browser. We choose Brave, rather than Chrome as done in related work, since its ad-blocking feature offers a more consistent browsing experience location-wise (see Fig. 7) which in turn implies offering a similar workload to the different devices. We invite the interest reader to leverage WPM's online service to compare the energy consumption of different browsers.

Table 4 summarizes the scale of this measurement in comparison with related work. We use Alexa's top 1,000 list from Nov. 2019 as an input. Out of the 1,000 websites we filtered 49 URLs potentially associated with adult websites—to avoid downloading and showing inappropriate content at our participating institutions—and 83 URLs associated with multiple top-level domains, e.g., `google.it` and `google.fr`. In the latter case, we kept the most popular domain according to Alexa, i.e., mostly the `.com` domain. Before the full experiment, we quickly tested the remainder URLs via a simple GET of the landing page. We find that 57 URLs had problems related with their certificate, 38 URLs timed out (30 s) and 58 URLs responded with some error code, 403 (Forbidden) and 503 (Service Unavailable) being the most popular errors. In the end, we are left with 715 active URLs.

We start by validating the assumption that Brave's ad-blocking helps in offering a similar workload to each device, irrespective of its location. Figure 9 shows the CDF of the bandwidth consumed across websites when accessed from the three different device/locations. Overall, the figure confirms our assumption showing minimal difference between the four CDFs. Note that some variations are still possible because of, for instance, OS-specific differences, geo-located

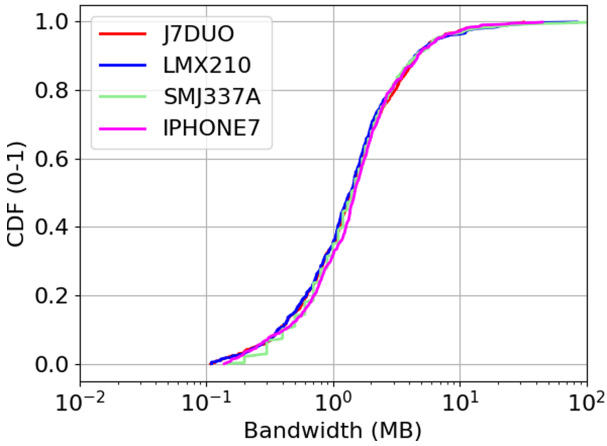

Fig. 9. CDF of bandwidth consumption.

content, or consent forms that tend to be more prevalent in Europe due to the General Data Protection Regulation (GDPR) [16].

Figure 10 shows the CDF of the energy (J) consumed across websites, measured on each device. The J7DUO is by far the most power hungry device consuming, on average, 50% more energy than both Android and the iOS devices—a trend that has previously been observed during benchmarking (see Fig. 6). A more similar trend is instead shared by the other devices, with most websites (∼80%) consuming between 10 and 20 J. The main differences can be observed in the tail of the distributions (10–20%) where the results start to diverge.

For completeness, we also analyze the CPU utilization during the test. We focus on Android only since, due to OS restrictions, it was not possible to obtain the CPU traces. We sample the CPU utilization every 3 s during each website load and report the 25th, 50th, and 75th percentile, respectively. Figure 10 summarizes this analysis as boxplots (across websites) for each percentile and device. An intuitive way to read this plot is to consider that low CPU values (25th percentile) refer to times before and after the CPU load, while the high CPU values (75th percentile) refer to times when the webpage was loaded. We observe that LMX210 and SMJ33, despite mounting similar hardware (see Table 2), exhibit different CPU usages, with SMJ33 spending overall more time at higher CPU utilization. The reason behind this are manifold, e.g., different OS versions and vendors. As expected, J7DUO suffers from less CPU pressure thank to its overall higher resources.

To conclude, BatteryLab allows to measure websites power consumption (and more) at unprecedented scale, in term of number of websites, devices, and network conditions. This opens up a new set of interesting research questions that we hope will appeal to the broader research community.

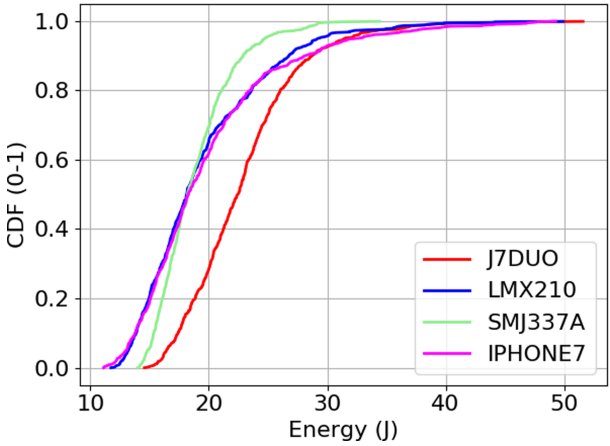

Fig. 10. CDF of power consumption.

6 Related Work

Several commercial products—such as AWS Device Farm [4], Microsoft App-Center [26], and Samsung Remote Test Labs [37]—could leverage BatteryLab's ideas to match our capabilities in a paid/centralized fashion. The same is true for startups like GreenSpector [20] and Mobile Enerlytics [27], which offer software-based battery testing on few devices.

In the research world, MONROE [1] is the only measurement platform sharing some similarities with BatteryLab. This is a platform for experimentation in operational mobile networks in Europe. MONROE currently has presence in 4 countries with 150 *nodes*, which are ad-hoc hardware configurations [28] designed for cellular measurements. BatteryLab is an orthogonal measurement platform to MONROE since it targets commercial devices (Android and iOS) and fine-grained battery measurements. The latter requires specific instrumentation (bulky power meters) that cannot be easily added to MONROE nodes, especially the mobile ones. Nevertheless, we are exploring BattOr [38], a portable power meter, to enhance BatteryLab with mobility support.

Last but not least, BatteryLab offers full access to test devices via a regular browser. This feature was inspired by [2], where the authors build a platform to grant access to an Android emulator via the browser to "crowdsource" human inputs for mobile apps. We leverage the same concept but also further extend it to actual devices and not emulators only. Further, remote access is just one tool in BatteryLab's toolbox and not our main contribution (Fig. 11).

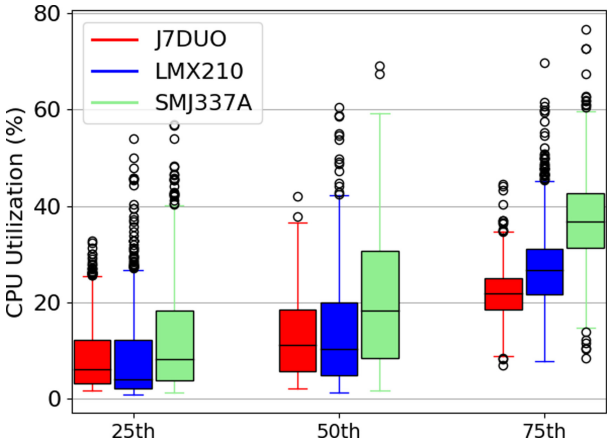

Fig. 11. Boxplot with percentiles of CPU usage.

7 Conclusion

This paper has presented BatteryLab, a collaborative platform for high-accuracy battery measurements where members contribute hardware resources (e.g., some phones and a power monitor) in exchange for access to the resources contributed by other platform members. To achieve this, we have built a complete prototyping suite which enables *remote power testing* for both Android and iOS devices. By releasing our code and setup, we invite the community to join BatteryLab, or at least we offer to eliminate the frustration associated with building "yet-another" home-grown performance measurement test-bed.

BatteryLab currently counts three vantage points, one in Europe and two in the US, hosting overall three Android devices and one iPhone 7. We evaluated BatteryLab with respect to its accuracy of battery readings, system performance, and platform heterogeneity. We show that BatteryLab's hardware and software have, for the most part, no impact on the accuracy of battery readings – when compared with a "local" setup. This is not true when visual remote access to the device is required, e.g., for *usability* testing. However, BatteryLab allows to "record and replay" usability tests which still offer accurate readings.

Towards the end of the paper, we also demonstrated how to design and run large scale measurements via BatteryLab. As an example, we have conducted, to the best of our knowledge, the largest scale measurement study of energy consumption on the Web, encompassing Alexa's top 1,000 websites measured from four devices on both Android and iOS. We have further released a web application integrated with BatteryLab which allows to measure the power consumption of a website, in real time.

Acknowledgment. This work was partially supported by the EPSRC Databox and DADA grants (EP/N028260/1, EP/R03351X/1).

References

1. Alay, Ö., et al.: Experience: an open platform for experimentation with commercial mobile broadband networks. In: Proceedings of the ACM MobiCom (2017)
2. Almeida, M., et al.: Chimp: crowdsourcing human inputs for mobile phones. In: Proceedings of the WWW (2018)
3. Amazon Inc.: Amazon Mechanical Turk (2022). https://www.mturk.com/
4. Amazon Inc.: AWS Device Farm (2022). https://aws.amazon.com/device-farm/
5. Amazon Inc.: Route 53 DNS (2022). https://aws.amazon.com/route53/
6. Appetize: Run native mobile apps in your browser (2022). https://appetize.io/
7. Apple Inc.: SharePlay (2021). https://developer.apple.com/shareplay/
8. Apple Inc.: How to AirPlay video and mirror your device's screen (2022). https://support.apple.com/HT204289
9. BatteryLab: Batterylab tutorial for new members (2022). https://batterylab.dev/tutorial/blab-tutorial.pdf
10. BatteryLab: The Web power monitor (2022). https://batterylab.dev/test-website.html
11. Bluetooth SIG Inc: Human Interface Device (HID) Profile (2022). https://www.bluetooth.com/specifications/profiles-overview/
12. BlueZ Project: BlueZ: Official Linux Bluetooth protocol stack (2022). http://www.bluez.org
13. Bui, D.H., Liu, Y., Kim, H., Shin, I., Zhao, F.: Rethinking energy-performance trade-off in mobile web page loading. In: Proceedings of the ACM MobiCom (2015)
14. Cao, Y., Nejati, J., Wajahat, M., Balasubramanian, A., Gandhi, A.: Deconstructing the energy consumption of the mobile page load. In: Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 1, no. 1, pp. 6:1–6:25 (2017)
15. Chen, X., Ding, N., Jindal, A., Hu, Y.C., Gupta, M., Vannithamby, R.: Smartphone energy drain in the wild: analysis and implications. In: Proceedings of the ACM SIGMETRICS (2015)
16. Data protection: Rules for the protection of personal data inside and outside the EU (2022). https://ec.europa.eu/info/law/law-topic/data-protection_en
17. Florian Draschbacher: RPiPlay - An open-source AirPlay mirroring server for the Raspberry Pi (2022). https://github.com/FD-/RPiPlay
18. Genymobile: Display and control your Android device (2022). https://github.com/Genymobile/scrcpy
19. Google Inc.: Android Debug Bridge (2022). https://developer.android.com/studio/command-line/adb
20. Greenspector: Test in the cloud with real mobile devices (2022). https://greenspector.com/en/
21. Hwang, C., et al.: Raven: perception-aware optimization of power consumption for mobile games. In: Proceedings of the ACM MobiCom (2017)
22. Jenkins: The leading open source automation server (2022). https://jenkins.io/
23. Let's Encrypt: A a free, automated, and open Certificate Authority (2022). https://letsencrypt.org
24. Leung, C., Ren, J., Choffnes, D., Wilson, C.: Should you use the app for that?: comparing the privacy implications of app- and web-based online services. In: Proceedings of the ACM IMC (2016)
25. Varvello, M., Katevas, K.: BatteryLab Source Code (2022). https://github.com/svarvel/batterylab

26. Microsoft, Visual Studio: App Center is mission control for apps (2022). https://appcenter.ms/sign-in
27. Mobile Enerlytics: The Leader in Automated App Testing Innovations to Reduce Battery Drain (2022). http://mobileenerlytics.com/
28. MONROE - H2022-ICT-11-2014: Measuring Mobile Broadband Networks in Europe (2022). https://www.monroe-project.eu/wp-content/uploads/2017/12/Deliverable-D2.2-Node-Deployment.pdf
29. Monsoon Solutions Inc.: High voltage power monitor (2022). https://www.msoon.com
30. Monsoon Solutions Inc.: Monsoon Power Monitor Python Library (2022). https://github.com/msoon/PyMonsoon
31. noVNC: A VNC client JavaScript library as well as an application built on top of that library (2022). https://novnc.com
32. Onwuzurike, L., De Cristofaro, E.: Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In: WiSec (2015)
33. ProtonVPN: High-speed Swiss VPN that safeguards your privacy (2022). https://protonvpn.com/
34. Raspberry Pi: Raspberry Pi 3 Model B+ (2022). https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
35. Ren, J., Rao, A., Lindorfer, M., Legout, A., Choffnes, D.: Recon: revealing and controlling PII leaks in mobile network traffic. In: MobiSys (2016)
36. RunThatApp: Enjoy Mobile Apps In The Browser (2022). https://runthatapp.com
37. Samsung: Remote Test Lab (2022). https://developer.samsung.com/remote-test-lab
38. Schulman, A., Schmid, T., Dutta, P., Spring, N.: Phone power monitoring with battor. In: Proceedings of the ACM MobiCom (2011)
39. TeamViewer GmbH.: TeamViewer (2022). https://www.teamviewer.com/
40. Thiagarajan, N., Aggarwal, G., Nicoara, A., Boneh, D., Singh, J.P.: Who killed my battery?: analyzing mobile browser energy consumption. In: Proceedings of WWW (2012)
41. TigerVNC: A high-performance, platform-neutral implementation of VNC (Virtual Network Computing) (2022). https://tigervnc.org
42. USB Implementers' Forum: Universal Serial Bus HID Usage Tables (2022). https://www.usb.org/document-library/hid-usage-tables-112
43. Mikhailov, V.: uhubctl - USB hub per-port power control (2022). https://github.com/mvp/uhubctl
44. Varvello, M., Katevas, K., Plesa, M., Haddadi, H., Livshits, B.: Batterylab, a distributed power monitoring platform for mobile devices. In: HotNets (2019)
45. Webpagetest: Test website performance (2022). https://www.webpagetest.org/
46. Wittenburg, P., Brugman, H., Russel, A., Klassmann, A., Sloetjes, H.: Elan: a professional framework for multimodality research. In: LREC, vol. 2006 (2006)