Experience: Developing a Usable Battery Drain Testing and Diagnostic Tool for the Mobile Industry

Abhilash Jindal
IIT Delhi and Mobile Enerlytics

Y. Charlie Hu
Purdue University and Mobile Enerlytics

ABSTRACT

In this paper, we report on our 6-year experience developing Eagle Tester (eTester for short) – a mobile battery drain testing and diagnostic tool. We show how eTester evolved from an "academic" prototype to a fully automated tool usable by the mobile industry.

We first present the design of our initial research prototype and discuss 8 key requirements for a usable battery drain testing and diagnostic tool gathered from some of the most popular software vendors in the Android ecosystem. These requirements posed interesting scientific and engineering challenges such as how to accurately estimate battery drain without requiring a priori power modeling, work on unmodified devices, and automatically monitor code evolution to generate high-fidelity battery spike alerts with actionable insights. These requirements motivated a complete overhaul of the eTester design and led to the creation of a novel battery drain testing methodology. We show how the redesigned eTester was used to effortlessly find battery bugs in some of the most popular Android apps with hundreds of millions of users, such as Netflix and CNN. We are open-sourcing eTester to encourage further research in battery diagnosis and to empower developers to write battery-efficient mobile software.

CCS CONCEPTS

• **Hardware → Power estimation and optimization**; • **Software and its engineering → Software maintenance tools**; • **Human-centered computing → Ubiquitous and mobile computing systems and tools**.

KEYWORDS

Mobile Device; Battery Drain; Testing and Diagnosis; Accuracy; Repeatability; Automated Testing; Actionable Insight

1 INTRODUCTION

In the last decade, there have been numerous efforts both from academia and industry to make mobile software energy-efficient. Academic efforts include characterizing and detecting energy bugs [15, 16, 23, 26], building power models of mobile hardware [4, 11, 25, 27], building fine-grained energy profilers [3, 24, 36], building energy-aware OS abstractions [29, 37, 38], energy-aware app adaptation [9, 19], and prescribing actionable diagnosis to app developers [14] and mobile users [22]. Industry efforts originated primarily from OS developers such as Google and Apple and from hardware manufacturers such as Intel and Qualcomm. These efforts include running automated static analysis checks to catch common energy bugs [45], providing battery drain diagnostic tools [40, 42, 52, 61, 65], reporting battery drain issues captured in the wild to developers [51, 58], creating energy-aware APIs for mobile app developers [41, 44] and increasing awareness of battery conscious software design through YouTube videos [48] and conference talks [59].

Tremendous progress has been made through these efforts on improving the battery life of smartphones. But we found from our interactions with hundreds of major players in the mobile software industry including popular apps like Netflix and Facebook, and OEMs like Google and Samsung, that their battery drain testing and diagnostic process continued to be ad hoc, unorganized or simply non-existent. This is because a battery drain testing and diagnostic tool (BTDT) usable by these developers did not exist.

In this paper, we report on our experiences in transforming an academic prototype Eprof into a fully automated BTDT tool, Eagle Tester, or eTesterfor short, usable by the mobile industry. This paper is divided into 5 parts. Section 2 provides the underlying principles and engineering design of our academic prototype Eprof. Eprof is a "textbook" battery drain profiler that extends a performance profiler such as gprof [10] to the energy dimension by breaking down the total battery drain of an app among software modules to identify battery drain "hotspots".

Section 3 describes our key learnings from our interactions with hundreds of major players in the mobile ecosystem. We surveyed various battery drain testing and diagnostic practices prevalent in the mobile software industry and gathered feedback about Eprof through giving live demos and doing deployments. We report the key gaps in battery drain testing and diagnostic practices and in existing BTDTs including Eprof as conveyed to us by these developers. They identified battery drain regression testing and diagnosis as an important missing piece and described 8 key requirements for a usable BTDT. Primarily, these developers require a prerelease BTDT tool that accurately estimates battery drain without requiring a priori power modeling, works on unmodified devices, and can keep automatically monitoring code

ACM MobiCom '21, Jan 31-Feb 4, 2022, New Orleans, LA, USA

ACM ISBN 978-1-4503-8342-4/22/01…$15.00
https://doi.org/10.1145/3447993.3483269

evolution to generate high-fidelity battery spike alerts with actionable insights.

Sections 4 - 6 describe the complete redesign of eTester from our academic prototype Eprof to address these key requirements for making eTester usable by the mobile industry. Section 9 discusses ongoing work of extending eTester to support 5G and AI/ML-based apps and to estimate battery drain without requiring a priori power modeling. Section 7 evaluates eTester's efficacy in fulfilling two quantitative industry requirements: accuracy and repeatability. Section 8 highlights a few energy bugs found by eTester in some of the most popular Android apps with hundreds of millions of users such as Netflix and CNN. Section 10 contrasts eTester with other BTDT tools developed by the mobile software industry.

Section 11 takes a step back to examine lessons learned from the 6 years of experience developing eTester. These lessons can be applied to developer tools and more generally to systems incubating out of research.

This paper makes the following contributions:

- It presents industry requirements gathered over a course of 6 years of industry interactions. These requirements posed interesting scientific and engineering challenges. We managed to solve all requirements except one: to make fine-grained BTDT seamlessly work without requiring a priori power modeling.
- It presents the drastic design changes we made to Eprof and demonstrates their efficacy in meeting the requirements.
- It describes a novel testing methodology for obtaining highly repeatable battery drain measurements even for short test runs and a statistics-based algorithm for generating high-fidelity battery drain regression alerts.
- It provides relevant lessons learned helpful to systems researchers who care for real-world adoption of their research prototypes.
- We have open-sourced eTester to empower developers to write battery-efficient software and to facilitate further research. eTester source code is available for download at https://bitbucket.org/mobileenerlytics/etester/.

2 THE PROTOTYPE

Our initial design of a BTDT was essentially a "textbook" energy profiler, Eprof [24].

2.1 Energy Profiling Basics

Energy profiling has been well studied [3, 8, 24, 30, 36, 39, 55]. Operationally, profiling the battery drain of an app in an app run consists of three tasks: (1) During the app run, *tracking app activities*, *e.g.,* the invocation of software modules at different granularities such as methods, threads, and processes, similarly as in conventional performance profilers such as gprof [10], by source code instrumentation or by runtime sampling of the execution stack, and by logging context switches of per-thread/per-process execution. (2) *Tracking power events* such as phone component usage, by logging all the power events of each phone component, typically via OS-exported logging facilities such as `atrace` in Android. (3) In offline processing, *performing energy accounting* by mapping the battery drain due to power events to app activities, using *power models*. Power models are mathematical functions that capture the correlation between power events that trigger the phone component usage and the resulting component power draw. The models need to be derived offline, typically using microbenchmarks, for each mobile device model. Using the power models, the total battery drain by the app run is attributed to the individual software modules by (i) feeding the power events per phone component to its power model to estimate the component power draw, and (ii) mapping the power draw and hence battery drain back to the software modules following the power event–app activity mapping derived in Steps (1) and (2).

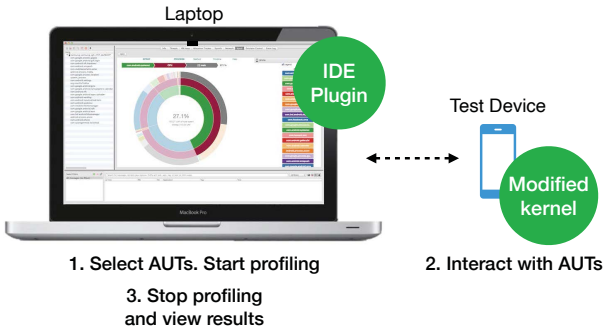

Figure 1: Eprof architecture and developer workflow.

2.2 Eprof Implementation

Figure 1 shows the architecture and developer workflow of Eprof. Eprof is developed as an IDE plugin with buttons to select the App Under Test (AUT), and to start and stop profiling. A developer connects her laptop via USB with a phone that is rooted and flashed with a custom Android kernel, and then clicks the start profiling button. The custom Android kernel is compiled with flags that allow tracing system calls.

Upon starting profiling, Eprof forks three loggers on the laptop to capture traces from the phone via the USB connection: (1) an `atrace` logger to log CPU frequency changes, thread-level context switches, and GPU frequency and state changes; (2) a network system call logger using `systemtap` [63]; and (3) an AUT method logger using Android's `traceview` [50]. These loggers stream traces from the phone while the user interacts with the AUT on the phone.

When the developer finishes her test, she clicks the stop profiling button. Eprof processes the captured traces to perform energy accounting following Step (3) described in Section 2.1 for CPU, GPU, WiFi and LTE.

3 MOBILE INDUSTRY REQUIREMENTS

We began our journey of transforming our academic prototype EPROF to a usable tool ETESTER by engaging with major players in the mobile software industry. Our goal was to understand the importance of battery drain testing and diagnosis, to learn about existing industry practices, and to gather industry requirements to improve our BTDT prototype. Some of these interactions are anonymized as they were done under a Non-Disclosure Agreement.

3.1 Battery Drain Testing and Diagnosis in the Mobile Industry

From our interactions with mobile software vendors, we learned that there is a significant variation in battery drain testing practices.

(1) No Testing. We started our interactions with small app developers working in teams of just 1 or 2 engineers. We quickly learned that such developers are focused on making their apps popular by adding viral app features. Battery efficiency is not a primary goal for them; they do not have any setup to test the battery drain of their apps.

Next, we focused our efforts towards popular app developers and OEM vendors. We learned of a few popular apps' developers who also did not have a battery drain testing setup because the daily average time spent by the users on their apps was less than 5 minutes; most of their users used the desktop version of their product. Popular Game app developers told us that their users expect the apps to drain a lot of battery while playing the games and hence they hardly received any complaints about battery drain. They just optimized the CPU and GPU resource consumption of their apps to make sure that the games remain responsive but did not have any setup to test battery drain.

Other than these exceptions, the rest of the industry had battery optimization setups.

(2) Black-box Testing, No Diagnosis. Many vendors had built in-house, often rudimentary, battery drain testing and occasionally diagnosis setups to prevent embarrassing battery drain problems. For example, two popular music streaming apps leave a test device playing music overnight and manually inspect the battery level drop next morning. Brave Browser [54] built BatteryLab [33] that connects the test device with a Monsoon power monitor [1] and allows measuring the total phone battery drain while running automated tests. Two major phone OEMs had a similar setup to run battery drain regression test on the Android framework running on their devices.

Such "black-box" testing does not provide any insight into the root cause of battery drain difference. Indeed, if the testing shows that a new software version drained much more battery than the previous version, the entire development team would enter a "fire-fighting" mode to debug why the battery drain increased, in an ad hoc manner, by looking at the code diff since the last test. This would sometimes delay app release.

(3) Diagnosis, No Testing. Battery drain diagnosis at Facebook was done using the Battery Metrics library [55]. To use Battery Metrics, a developer manually instruments her app with library calls and releases the app. When this app runs on devices in the wild, the library will collect resource usage information as per the instrumentation. This setup was able to catch several battery regressions in the wild after new releases were rolled out to an initial set of users – a common practice by app vendors. However, these regressions were getting caught *after* impacting the initial set of users.

Table 1: Industry requirements for a usable battery drain testing and diagnostic tool.

Requirement	Vendor type	Perceived Severity
Test Setup		
S1: Automated testing	Music streaming app, Social network app	Critical
S2: Runs on unmodified devices	Music streaming app, Mobile cloud testing company	Important
S3: No a priori power modeling	Popular OEM	Important
Usability		
U1: Short test runs	Music streaming app	Important
U2: Long test runs	Location Intelligence app	Critical
U3: Actionable insights	Mobile cloud testing company	Desirable
Fidelity		
F1: Accurate	Browser app	Critical
F2: High-fidelity alerts	Social network app	Critical

3.2 Industry Requirements for BTDT

We demonstrated our academic prototype EPROF to mobile software developers and gathered their requirements. Table 1 summarizes the requirements which are grouped into three categories–tool setup, usability, and fidelity. The table further shows for each requirement, the vendors who are most concerned about the requirement and the perceived severity level. Among the three severity levels, "Critical" requirements are a show-stopper if not supported for most users, "Important" requirements, if not supported, negatively affects the usage decision for some users, and "Desirable" requirements are strongly preferred but users will still use the BTDT without it.

3.2.1 Setup Requirements. The first set of requirements are all concerned with the setup of the tool, some of which are viewed by developers as "show-stoppers" if not supported.

S1: Automated testing. Most development teams do not have a dedicated battery team, and thus running manual energy profiling, such as in EPROF, is disruptive to developers' workflow. For this reason, lack of support for testing automation is considered a "show-stopper" by most of the vendors who are interested in testing their software's battery drain. In particular, the BTDT tool needs to support measuring the battery drain through automated user interaction scripts. These automation scripts can be black-box tests since app developers do not have access to the competitor apps' source code but are keen to do competitive analysis.

S2: Runs on unmodified devices. EPROF requires root access and a custom kernel to log system calls and other power event triggers. These permissions can be granted by rooting the phone and flashing a custom kernel. However, Popular OEM refused to share their device kernel with us due to security reasons. They further barred us from rooting their test devices since some kernel processes do not run on rooted devices and hence battery drain measurements on rooted devices may be unreliable. A few app developers are apprehensive of rooting their devices since rooting nulls the warranty of the devices and if not done correctly can "brick" the device, *i.e.,* create irreparable damage to the device such that the device will not even boot up. Other app developers, especially the ones developing enterprise apps like Facebook, are comfortable with rooting their test devices. This poses an important requirement that the BTDT tool needs to run on unmodified devices.

S3: No a priori power modeling. EPROF could support a wide range of devices but requires gaining access to the device, bypassing its battery to connect the Monsoon power monitor, and running extensive microbenchmarks to create a per-component power model. However, OEMs often need to test their new devices yet to be released to the market, and hence cannot make them accessible to us first which is needed to create the power models used by EPROF. ETESTER thus needs to do battery drain estimations and accounting for devices that are not accessible to us for creating its power model. This requirement is critical to OEM vendors but not as critical to app vendors who typically test app battery drain on a few representative phones, *e.g.,* from recent phone generations.

3.2.2 Usability Requirements. The second set of requirements are concerned with the usability of the BTDT tool.

U1: Short test runs. It should be possible to test a single user flow in an app, *e.g.,* search for a friend in an IM app and send the friend a voice message, which may not even produce a 1% battery level drop. Short test runs allow for rapid continuous testing and help direct battery drain issues to the team lead responsible for that user flow. This requirement is deemed important by vendors of many complex apps and OEMs which typically involve large development teams. Smaller teams, within the large development teams, are typically responsible for the development and testing of separate modules.

U2: Long test runs. It should also be possible to run long tests to test the background battery drain behavior and to test the battery drained in supporting long user interactions such as watching a movie. This is not possible with EPROF since it generates too many logs, *e.g.,* at the rate of 6 Mbps while watching a Youtube video. Since testing background activities is of interest to many apps and OEMs, this requirement is considered critical.

U3: Actionable insights. EPROF accounts the total energy consumption of an app process to its Java method calls. In talking with developers, we found this "textbook" approach of energy accounting to be often insufficient in providing actionable insights due to the following reasons: (1) Method-wise energy breakdown could not be done effectively for phone components such as screen and hardware decoder; (2) Top battery draining methods are often Android framework or third-party library methods, making method-wise breakdown not actionable for app developers; (3) Software not written in Java, such as React Native apps and kernel workers, could not benefit from EPROF's energy breakdown into Java methods; and (4) OEM developers debugging whole system issues and app developers debugging app interactions with system services [2] could not gain much insights from just one app's method-wise energy breakdown. As such, the ability to output actionable insights that can simplify or automate the process of debugging battery drain spike of a new app version is considered a highly desirable feature of a BTDT tool.

3.2.3 Requirements for Fidelity of Testing Output . The third set of requirements are concerned with the tool's fidelity.

F1: Accurate. To generate high-fidelity testing and diagnosis output, the BTDT tool needs to output highly accurate battery drain estimation relative to the ground-truth measurement. This is perceived as a critical requirement of a BTDT tool by all vendor types.

F2: High-fidelity alerts. Since most developer teams do not have a dedicated battery team, the BTDT tool must generate alerts whenever the latest code changes cause a spike in battery drain. These alerts must have high-fidelity for developers to have continued confidence in them. If the alerts have many false positives, developers would start ignoring them, making the test setup ineffective [18]. If the alerts have many false negatives, then battery drain problems will creep into releases, again making the test setup ineffective. For this reason, the ability to generate high fidelity alerts is viewed by all vendor types as a critical requirement.

We note the subtle but important difference between F1 and F2. F1 is about the accuracy of a particular test run, that the BTDT tool should report accurate battery drain profiling results. However, even if a BTDT is accurate for each test run, the battery drain can potentially vary across different test runs as it is affected by many factors (see §6.2). Hence, F2 is (1) about repeatability, *i.e.,* how to reduce the variations in battery drain results across test runs, and (2) about dealing with non-repeatable behavior, *i.e.,* confidently

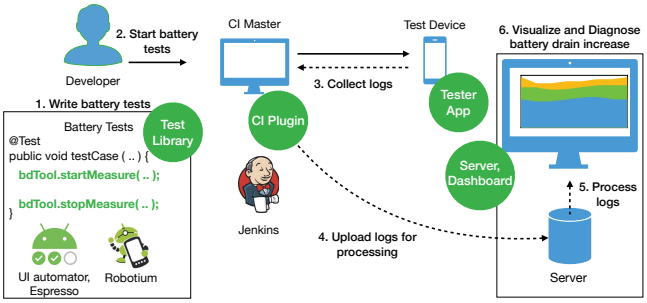

Figure 2: eTester architecture and developer workflow.

flagging battery spikes when battery drain results from even identical test runs may differ from each other.

To meet the above requirements, we made a number of drastic changes to EPROF and transformed it into a new BTDT solution, ETESTER. We discuss the details of these design changes in the following sections.

4 SETUP IMPROVEMENTS

Below we describe the challenges in and our approach to addressing these setup requirements.

S1: Automated testing. To support testing automation, we completely revamped the IDE plugin architecture of EPROF to make ETESTER seamlessly integrate with Continuous Integration (CI) platforms as shown in Figure 2. This new architecture works through the following steps.

(1) Write battery tests. Developers start with writing automated battery drain tests using standard Android testing libraries: white-box testing using Espresso [43] and black-box testing using UI Automator or Robotium [62]. The tasks for which battery drain needs to be measured are wrapped with two simple methods provided by ETESTER *test library*: `startMeasure` and `stopMeasure`. To test battery drain of background tasks, automated tests can either just sleep for some time, waiting for background tasks to happen, or emulate external events that trigger background activities, *e.g.,* sending an email to the device under test.

(2) Start battery tests. After battery drain tests are committed, these tests are run by the CI setup such as Jenkins on a connected test device according to a cadence defined by the CI setup such as "run at each code commit", "run at each pull request", or "run after every N hours".

(3) Collect logs. When the test execution reaches the `startMeasure` method call, the ETESTER test library sends an IPC message to the ETESTER *app* installed on the test device. The ETESTER app starts capturing logs and writes them to the device's SDCard. When the test execution reaches the `stopMeasure` method call, the test library sends another IPC message to the ETESTER app to stop logging.

(4) Upload logs for processing. When all the tests are finished, ETESTER *CI plugin* uploads all the logs present on the device to the ETESTER server using a REST API.

Table 2: eTester whole-system loggers using standard Android interfaces.

Interface	Logging purpose
atrace	CPU, GPU, and hardware decoder power estimation; thread-level CPU energy accounting from context switches
ps	For identifying process and thread names from their ids
screenrecord	Screen power estimation; correlating power timeline with screen updates
tcpdump	Network power estimation; socket-level network energy accounting
logcat	Correlating power timeline with system logs
dumpsys	GPS power estimation; correlating power timeline with system events such as alarms, location requests, and wakelocks
strace	Correlating power timeline with system calls

(5) Process logs. The ETESTER server processes the uploaded logs in a manner similarly as in an energy profiler (§2.1) to compute per-component instantaneous power consumption. Our novel high fidelity alert generation algorithm (§6.2.3) further compares the battery drain by the test run with that of a reference test run set by the user, *e.g.,* a previous app version or a competitor's app, and sends an email alert if the battery drain ratio is higher than a threshold.

(6) Visualize and diagnose battery drain increase. The developer can log in to the ETESTER *dashboard* to visualize and diagnose the battery drain measurements.

S2: Runs on unmodified devices. EPROF required flashing a custom kernel on the test device as it uses non-standard interfaces like `systemtap` to extract logs. To make ETESTER run on devices without a custom kernel, we decided to only use standard Android interfaces that do not require flashing a custom kernel, as shown in Table 2. We had to make significant modifications to EPROF to make this transition, *e.g.,* using packet based network power estimation instead of system call based estimation.

This removes the custom kernel requirement but still requires us to root the device This is because apps, including the ETESTER app, are not allowed to log data from some of these standard interfaces without rooting the phone. We observed that these logs, however, are available for reading through `Android Debug Bridge (adb)` which runs as a special Linux user `debug`. This `debug` user has higher permissions than the apps installed via Google Play Store but has lesser permissions than the `root` user. Thus, to further remove the rooting requirement, we ship a standalone Java application to perform logging, as an alternative to the ETESTER app, which can be pushed directly to the test device and run as the `debug` user.

Unfortunately in the current state of Android, this standalone Java application can only support a limited number of hardware components such as CPU, screen, GPS, WiFi and LTE. This is because for some phone components, *e.g.,*

GPU and Hardware Decoder, even the `debug` user is not allowed to read their logs. These logs are outside the control of Android and are logged in differing ways with differing permission models by the OEMs. **S2: Runs on unmodified devices** is thus only fulfilled to a limited extent; to extract full functionality from eTester, the developers still need to root their device. Achieving full eTester functionality on unmodified phones requires further logging standardization and enforcement from the Android ecosystem, a goal also shared by Android Perfetto [47].

Finally, **S3: No a priori power modeling** remains an unsolved challenge in eTester.

5 USABILITY IMPROVEMENTS

U1: Short test runs. This requirement is naturally met via testing automation described in §4 and via fine-grained energy accounting in eTester. We next discuss eTester's approach to address the remaining two usability requirements.

U2: Long test runs. To reduce the size of generated logs, we engineered two main changes to the Eprof loggers: (1) recording a lower resolution screen video, and (2) optimizing reading `atrace`. `atrace` is a ring buffer that holds traces for system events, such as context switches and CPU/GPU frequency changes. We observed that the logs from thread-level context switches in `atrace` form the major contributor to the large size of logs generated by Eprof. To reduce log size, we perform log pruning and compression on the fly before writing them to SDCard. We changed our logger to use multiple CPU cores available on modern smartphones. eTester starts one reader thread that reads logs from `atrace` and `(number of cores - 1)` writer threads that compress and prune the logs before writing them to SDcard. Co-ordination between the reader and writer threads is managed by a memory buffer pool system. Using these mechanisms, eTester generates 6 times less logs than Eprof, which allows longer running tests. eTester also allows selectively disabling logging of components, such as network and screen, to further reduce the size of generated logs.

U3: Actionable insights. As discussed in §3.2.2, Eprof's textbook approach to energy accounting cannot provide many developers, such as OEM developers and React Native developers, with actionable insights.

Our enhanced actionable insights in eTester were motivated by the key observation that modern apps heavily rely on system services provided by the mobile OS and the framework whose battery drain are not captured by method-level energy accounting. Accordingly, we focused the actionable insight design of eTester on correlating system-wide events with per-component power consumption as shown in Figure 3. (1) eTester output displays a power consumption timeline for each phone component synchronized with the recorded screen video, system logs from logcat, system call logs from strace, and system events from Batterystats which include events such as phone calls and audio playback. This allows the developer to easily identify major causes of power spikes in the timeline. (2) eTester also shows per-process power consumption for all components, per-thread CPU power consumption and per-socket network power consumption to provide further insights into the system power consumption. Together, these system-wide power diagnosis features in eTester have proved successful in finding bugs in popular apps as we will see in §8. We acknowledge that additional diagnosis features, *e.g.,* those specialized to other major phone components such as the wireless NIC or OLED display, can be developed, and we continue to evolve eTester to generate more actionable outputs.

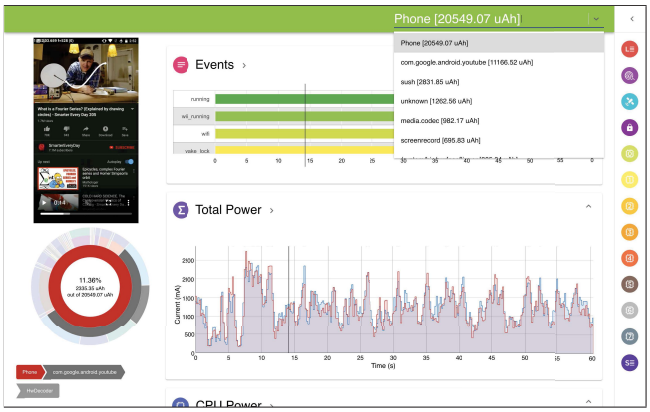

Figure 3: Actionable insights by eTester: (1) A variety of system events correlated with per-component power draw timelines; (2) per-thread and per-socket power draw timeline.

6 FIDELITY IMPROVEMENTS

6.1 Accurate battery drain estimation

One of the strengths of Eprof design is its high battery drain estimation accuracy from using accurate component-wise power models of the test device (§2). eTester inherits this high battery drain estimation accuracy from retaining the same energy accounting design to meet **F1: Accurate**. However, Eprof only supported a limited number of components commonly found in the literature, including CPU, GPU, GPS, and WiFi/4G NIC. eTester extends the power-hungry component coverage to the whole phone by including two additional components, OLED and hardware decoder, which are challenging to model.[1]

We developed an accurate OLED power model that significantly improves the accuracy of prior-art linear regression-based OLED power models [6, 17, 20]. Our new model divides up the RGB color space into smaller subspaces and develops linear regression-based power models for them individually. The resulting new model can estimate the OLED display power draw within just 3.3% of the ground truth, measured by a Monsoon power monitor, in displaying a set of 100 diverse images on Nexus 6 and Pixel 2.

[1] eTester also models the LCD display energy drain, which has become much less common in modern phones.

We also added to eTester accurate, fine-grained power modeling and battery drain accounting of hardware decoder which is widely used in video streaming apps. We carefully analyzed the timings of all the events related to hardware decoder in the `atrace` log and the timings of power change in the power monitor readings and found a strong correlation between MDSS (Mobile Display Subsystem) events and the spikes in the power reading. We then used MDSS events as the power events for hardware decoder's power model derivation and use. To our knowledge, this is the first known empirical power model for hardware decoder for smartphones. The model achieves a low average estimation error of 5.0% when compared to the Monsoon power monitor reading as the ground truth in replaying downloaded Youtube videos of varying resolutions including 1080p, 720p, 480p, 360p, 240p and 144p on Pixel 2.

6.2 High-fidelity alerts

Industry requirement for generating **F2: High-fidelity alerts** is in general at odds with supporting **U1: Short test runs**. This is because short tests have less scope of noise amortization and tend to have greater relative variations due to small total battery drain. To make measurements highly repeatable, we first enumerated reasons why a battery drain testing setup exhibits variations and then addressed them in our novel test methodology. We then developed a statistics-based algorithm that uses the battery drain estimations collected using the test methodology to compare a test run with a reference test run configured by the user, *e.g.,* last release of their software or a competitor's software, to generate high fidelity alerts.

6.2.1 Challenges for Repeatable Battery Drain Testing. Repeatable battery drain testing of mobile software is challenging because the battery drain is affected by many factors that can potentially vary between different test runs. We group these factors into three broad categories:

Phone internal variations. Modern phones are built with advanced hardware and run full-fledged time-sharing operating systems. As such, the behavior of mobile software and hence its battery drain can be affected by the complex interactions with other processes, the OS, and the hardware. *(1) Background apps and system processes* can contribute to the total power draw of the phone; *(2) CPU scheduler* can assign the same thread to cores of different sizes (big/LITTLE) running on different frequencies in different runs; and *(3) High temperature* can cause the CPU power to increase by up to 15% while operating under the same frequency and can cause thermal throttling [64].

Phone external variations. Popular mobile apps interact with users via rich UI and with remote servers over the network. As such, their battery drain can be affected by several external conditions. *(1) User interaction* variations lead to different app execution behavior; *(2) Wireless signal strength* affects both the latency and bandwidth of network data transfers and hence causes the wireless NIC to draw power of different magnitudes and for different durations; and *(3) Server/network latency and bandwidth* variations may change the app behavior, *e.g.,* triggering adaptive bitrate (ABR) in video streaming apps to switch to a lower video bit rate.

App state. Mobile software is complex, often containing features and internal optimizations that cause non-repeatable execution. Such execution variations can cause battery drain variations. *(1) Different starting states* trigger different app behavior, *e.g.,* a browser has to download the actual web page from the network if it is not in the cache, but can simply use it from the cache if present; *(2) A/B testing* may cause the same app binary to exhibit different behaviors; and *(3) Ads* can be dynamic in nature; displaying and downloading dynamic content can cause battery drain to vary among test runs.

6.2.2 Addressing Sources of Battery Measurement Variations. eTester automatically handles variations stemming from phone-internal conditions and from a few phone-external conditions and app states.

- *Fine-grained energy accounting* in eTester, borrowed from Eprof, accurately calculates the portion of the phone battery drain that is due to each process and thread under test and thus can remove the battery drain due to background apps including the eTester app itself and system processes.
- The eTester app *monitors phone temperature* and ensures that it is at normal temperature before each test run to avoid variations due to the thermal effect.
- To ensure consistent user interactions across different test runs, eTester supports *automated testing.*
- eTester automatically *cleans app caches* before each test run to ensure the same start state.

To control the remaining sources of variations, we recommend that QA engineers follow additional testing guidelines:

- *Stabilize external conditions.* For tests that use the network, place the testing phone at a location that has stable wireless signal strength (WiFi or cellular) and if possible use a private WiFi AP. We further recommend using an in-house server or a mocking server (*e.g.,* using a proxy) to control server response time.
- *Control app states.* The engineer performing the battery drain test should make sure that the same A/B testing version of the app is used across multiple test runs. We also recommend using an ad-free app version, *e.g.,* using a premium account, to remove battery drain variations because of ads.
- *Repeat test runs.* The CI setup is recommended to be configured so that *each test is performed at least 5 times.* This is done to address uncontrollable variations, such as from the CPU scheduler.

6.2.3 Generating High-Fidelity Battery Drain Spike Alerts. After addressing the sources of variations, we receive 5 or more

battery drain measurements for each test run. For generating an alert for a test run, we need to find out the ratio of the average battery drain by that test run with the average battery drain by a reference test run set by the user, *e.g.,* last release of their software or a competitor's software. Simply taking the average of the respective measurements and then taking the ratio of these two averages is unsound since it does not account for the variance and the number of samples in the measurements.

To compute this ratio with a high degree of confidence, we adopt a statistics based approach. We denote the battery drain by the test run as B and the battery drain by the reference test run as R. We further assume that both B and R are normally distributed random variables, *i.e.,* $B \sim N(\mu_B, \sigma_B^2)$ and $R \sim N(\mu_R, \sigma_R^2)$, and the battery drain measurements are samples from these distributions. We need to find the ratio of μ_B/μ_R and generate an alert if this ratio is greater than a user defined threshold τ.

Finding the ratio μ_B/μ_R is challenging since the actual B and R distributions are unknown to us; we only have access to the samples. Moreover, B and R can have unequal number of samples since test runs can crash or users might re-configure the number of times that each test case is run. B and R can also have unequal means and unequal variances since they are measured independently and are battery drain measurements of different code versions of the same software or of two completely different software.

To our best knowledge, there is no direct mathematical expression for estimating the ratio of the averages of two normal distributions from their samples. To judge whether to generate an alert and estimate the precise ratio μ_B/μ_R, we developed a novel iterative algorithm. The algorithm makes use of Welch's unequal variance t-test [34], a statistical test used to determine whether there is a significant difference between the means of two normal distributions. The t-test expressed as $P(X > Y) \geq 0.95$ verifies with 95% confidence that the mean of X is greater than the mean of Y. The test can operate on the samples from two normal distributions; the actual distributions are unknown.

Our algorithm takes a threshold τ and confidence level α as input. It first finds if $P(B > \tau R) \geq \alpha$ using Welch's t-test. To set up this t-test, we simply multiply the samples from $R : R_1, R_2, ...$ with τ; since R is normally distributed, τR is also normally distributed. If the t-test fails, it does not generate an alert; otherwise it generates an alert, containing the precise ratio μ_B/μ_R, to notify users of a battery drain spike. This precise ratio is calculated as follows. First, the algorithm repeatedly doubles hi starting from τ until $(P > hi \cdot R) \geq \alpha$ fails. This gives a ratio $hi/2$ at which t-test succeeds and a ratio hi at which t-test fails. Next, it performs a binary search between $hi/2$ and hi to find the highest ratio γ that satisfies the t-test with the given confidence level α, *i.e.,* $P(B > \gamma R) \geq \alpha$. This γ gives the precise ratio of μ_B/μ_R.

To summarize, eTester first makes battery drain measurements highly repeatable in the presence of short test runs by carefully mitigating sources of variations and then generates high-fidelity alerts using a statistics-based algorithm.

7 VALIDATION

In eTester design, we have addressed all of the industry requirements outlined in §3 except **S3: No a priori power modeling**. In this section, we evaluate the quantitative guarantees of eTester: accuracy and repeatability of battery drain measurements. We omit overhead evaluation because eTester performs accurate fine-grained energy accounting and can accurately isolate and remove the energy overhead due to itself [2]. We perform the evaluation using 6 common test scenarios on 4 popular pre-installed Google apps as summarized in Table 3. All test scenarios consume power on OLED screen, and the 2 video playback test scenarios from YouTube app also drive hardware decoder. All tests are performed using automated scripts written using UI Automator and are run 5 times each for a total of 30 runs. Before each test run, eTester clears the local device cache. In the case of Youtube, our automated tests also clear the saved user account history before each test run and we use a premium account to disable ads. The tests are performed on a Nexus 6 phone that is connected to a stable WiFi connection with the screen brightness level fixed to 38%; the brightness level was picked by Android's adaptive screen brightness for our indoor environment.

7.1 Accuracy

To measure the battery drain error for a given test duration, we compare the total estimated battery drain with the ground truth battery drain for the duration. The total estimated battery drain is obtained by integrating the instantaneous power estimations from eTester over the duration, and the ground truth battery drain is obtained by integrating the power readings from the built-in current sensor (using the Android power sensor API which has a sampling resolution of 170ms on Nexus 6).

We first evaluate the end-to-end energy error for the 30 test runs. We found that all 30 runs have less than 6.5% end-to-end energy error. Table 3 shows the average end-to-end error for the 6 test scenarios.

We further evaluate fine-grained energy estimation error of eTester. Since eTester uses accurate power models to estimate the power draw of all major phone components and since eTester logging logs all the power events of those components (see Table 2), eTester can estimate the instantaneous power draw of the phone components. To measure the fine-grained energy estimation error, we choose 2 seconds as the time granularity as we learned from our interactions with developers that they rarely care to debug power spikes that last less than 2 seconds. We found that for the time-granularity of interest to eTester, 2 seconds, the energy reported from the built-in current sensor is very similar to that from a Monsoon power monitor. We therefore use the built-in current sensor for fine-grained accuracy testing. We have also added the current sensor reading along side eTester's power timeline

[2]For this reason, the mobile software vendors we interacted with generally find the overhead of eTester not a concern.

Table 3: List of apps tested, usage scenarios, accuracy, and repeatability metrics including average end-to-end error, coefficient of variation, and average pair-wise component and thread Manhattan distances (MD).

App	Test scenario	Test duration	Average end-to-end error	Coefficient of variation	Component MD	Thread MD
Google Calendar (6.0)	Navigating, creating and browsing scheduled events	45s	1.7%	3.3%	5.2%	1.2%
Chrome (75.0)	Load and scroll a webpage	35s	0.5%	5.3%	10.5%	7.6%
Google News (5.12)	Browsing, reading news from The Verge	1m45s	2.4%	1.8%	4.2%	1.7%
YouTube (14.25)	Browsing	35s	2.8%	3.5%	5.9%	1.7%
YouTube (14.31)	Search, landscape video playback	1m	2.6%	2.5%	3.6%	2.1%
YouTube (14.31)	Search, portrait video playback	1m	4.6%	1.4%	3.1%	0.7%

output so that developers can perform the same accuracy testing as reported in this paper.

To measure fine-grained battery drain estimation error of eTester, we break the total test duration for the test run with the median end-to-end battery drain error out of the 30 test runs into 2-second windows. This test run was a navigation test of the Google Calendar app. We found that about 80% of the 2-second windows have an absolute error less than 10% and 90% of the windows have error less than 15%. This shows that the fine-grained battery drain estimations by eTester are within a small margin of the ground truth power draw.

7.2 Repeatability

The first metric we track for the overall repeatability of eTester's testing results is the co-efficient of variation (CV). Formally, CV is the ratio of the standard deviation to the mean; the lower the value of CV, the less variation is in the data, *i.e.,* the experiments are more repeatable. To our best knowledge, CV has never been measured for battery drain testing in the past [3, 24, 36, 40, 42, 52, 61, 65], but statistically a CV of less than 10% is considered "very good" [7]. Table 3 shows that eTester's CV ranges from 1% to 6%, confirming that eTester test outputs are highly repeatable. As expected, the CVs are relatively higher for shorter test runs.

Even though the CV is low, the breakdown of battery drain into threads and phone components may still be different across runs. To evaluate the repeatability of battery drain breakdowns, we use a methodology similar to the Google data center profiler [28]. We calculate the pair-wise component and thread Manhattan distances (MD). The Component MD $C(X, Y)$ between two test run outputs X and Y is defined as

$$C(X,Y) = \sum_{c \in CPU,GPU,SCREEN,..} |p_X(c) - p_Y(c)|$$

where $p_X(c)$ and $p_Y(c)$ are the percentages of energy consumed by component c in test runs X and Y, respectively. Low component and thread MDs between two test runs signify stable component and thread breakdowns of the total battery drain. Table 3 shows that the average pair-wise component MD for all scenarios ranges from 3.1% to 10.5% and the average pair-wise thread MD ranges from 0.7% to 7.6%. This shows that both the component energy breakdown and the thread energy breakdown output by eTester are repeatable across test runs.

8 CASE STUDIES

8.1 Netflix

By April 2019, there had been a year-long mysterious energy leak in the Netflix app with thousands of user complaints on the app's Google Play page. eTester was used to understand this energy leak by emulating CI-driven battery drain testing of Netflix app versions after the energy leak (version 6.4.0) and before the energy leak (version 6.1.0) on a Nexus 6 device running Android version 6.0.1. In particular, for each version of the app, eTester performed a 1-minute idle test; during the test, the app simply sat idle in the background with no user interactions and no video streaming. eTester showed that while version 6.1.0 drains only up to 2 mA of current, thread `ProcessManager` in version 6.4.0 drains a relatively constant current of 300 mA on the CPU!

Digging further, the developer looked at eTester's strace tab for the `ProcessManager` thread and found that the thread is repeatedly calling the `wait4` system call with each call returning an `ECHILD` error (no child processes). The `wait4` system call is used to wait for child processes to exit. Netflix fixed this bug in version 7.8.0.

8.2 Xfinity Stream vs. CNN

eTester was used to run a competitive test between Xfinity Stream version 4.11 and CNN version 5.16. The test streams identical TV content for 3 minutes using each app. Both apps were tested on a Nexus 6 phone connected to a stable WiFi. eTester showed that the CNN test consumes 27% higher energy than the Xfinity Stream test and further showed that this energy difference stems primarily from the `/system/bin/logd` process. This process is responsible for ingesting all the log messages produced in the system, making them viewable from Android's `logcat`.

The developer next focused his attention to the Logcat tab of eTester's timeline view. This tab reveals that the

CNN app logged over four times more logs – 8685 lines of log messages compared to 1944 logged by the Xfinity Stream app! It further showed that out of the 8685 log lines, 7920 lines are "Debug" log messages. CNN developers had accidentally left "Debug" logs enabled in a released app, defying Android's guideline of silencing them [46].

9 DISCUSSIONS AND ONGOING WORK

We discuss ongoing work to support the last requirement not currently addressed and several new issues that will rise as smartphones and mobile apps continue to evolve.

S3: No a priori power modeling. As discussed in §3.2, the primary challenge in meeting usability requirement **S3: No a priori power modeling** is to accurately estimate and account battery drain for devices not accessible to us for offline power modeling. In our ongoing work, we are exploring self-constructive online power model derivation (SPMD) which will omit the need to derive the power models for each handset before the testing time. Previously, SPMD was explored in [5] to improve the granularity of total energy estimation of an app to be finer than power sensor reading. In contrast, we are exploring SPMD to generate component-wise power models in an online manner.

The basic idea of SPMD is log the instantaneous total phone power draw or energy drain using the built-in power sensor (*e.g.,* via Android APIs) in addition to logging all the power events as before, *e.g.,* CPU utilization at each frequency, during each app test run. In post-processing, it solves a system of equations that express the total phone energy drain as linear functions of the component power draws (unknowns).

The challenge in making SPMD to work, however, is whether there are enough diversity in the system of equations, so that the regression solver can output meaningful solutions as the component-wise power model parameters.

Supporting new phone components. eTester can be easily extended to support new hardware components. For example, newer wireless network interfaces such as 5G mmWave can be supported by plugging in 5G power models, *e.g.,* [21, 35], and new special processors such as NPU and TPU [12, 13, 32] can be supported by developing and adding corresponding power models.

Supporting new classes of power-hungry apps. eTester can readily or be easily extended to support newer classes of power-hungry apps. If the new class of apps are using existing hardware components, no change is needed. If they use newer hardware components, *e.g.,* AI/ML-based apps such as Camera, Socratic [31], and Siri and Google Assistant (running offline) which represent a new generation of power-hungry apps either use the mobile GPU which is already supported by eTester, or use special processors such as NPU and TPU [12, 13, 32] which can be easily supported by eTester.

10 RELATED WORK

We already reviewed academic research related to energy profiling in §2. In this section, we focus on mobile software optimization tools developed concurrently with eTester in the industry. We discuss them under three categories: developer tools, pre-release tools, and post-release tools. We contrast them with eTester on how they compare on the 8 key requirements. We acknowledge that these tools may provide functionalities outside of the scope of a usable BTDT.

Developer tools. Lint checks [45] catch known bug patterns using static analysis such as unreleased wakelocks and frequent alarms. Although effective in improving mobile software, these checks do not catch arbitrary battery drain regressions.

Developers also have access to several resource profilers [49, 61] and system tracers [47]. None of these tools measure or find the root cause for battery drain. Android Energy profiler [42] shows trends in power consumption by labeling them as "Light", "Medium", and "Heavy"; it neither is automated, nor provides quantitative battery drain breakdown to effectively catch battery drain regressions. Similar to eTester, AT&T Video Optimizer [52] uses power models to estimate battery drain. Video Optimizer's power models extensively capture network power behavior but do not capture the power behavior of other smartphone components such as OLED screen and heterogeneous CPUs [53] and thus can not provide accurate battery drain estimations.

Pre-release tools. Our interactions with the smartphone software industry taught us that it has a strong inclination towards automated testing tools over manual developer tools. Indeed, there are a number of popular pre-release testing tools testing different aspects of software development such as UI responsiveness [60], app performance [56], and network performance [57]. Battery drain testing setup, BatteryLab [33], was discussed in §3.1. None of these tools perform detailed battery drain diagnostics or generate battery drain spike alerts. eTester is a pre-release battery drain testing and diagnostic tool that gives accurate battery drain measurements and generates high-fidelity battery drain regression alerts.

Post-release tools. Android app publishers can get insights into their app's behavior in the wild using Android Vitals [51]. Developers can also ask their users to send them bug reports that can be analyzed with Battery Historian [40]. Battery Metrics SDK from Facebook was discussed in §3.1. Unlike these tools, eTester is a pre-release BTDT that can capture battery drain regressions *before* negatively impacting users' phone battery.

11 LESSONS LEARNED

Our efforts in pitching Eprof in industry conferences, giving on-site demos, and performing deployments taught us many valuable lessons. In this section, we focus only on the lessons that highlight the somewhat unexpected gaps between academic research and industry practices: these are

the concepts that we personally never considered as academics but turned out to be critical for adoption in the industry.

Ease of setup and use. 6 out of the 8 industry requirements in our study are related to setup and usability. As academics, we often overlooked these concerns. For example, we took modifying the Android kernel and framework for granted, assuming that as long as EPROF could provide a good number of insights, the extra manual work of modifying devices must be acceptable. But for EPROF, this turned out to be the biggest barrier to entry into the industry. Human hours are a valuable resource in the industry; building fully automated, intuitive, and easy to setup systems is crucial for industry adoption.

Fulfilling the needs of different personas. While developing EPROF, we completely focused our efforts on developers who routinely debug battery drain problems in their software. From our fruitless initial pitching efforts, we learned that the features that appealed to developers actually did not appeal to other personas in our adopters; these personas were often also the decision makers for the deployment of the tool. Hence, we had to first understand the motivations of these different personas and then build a number of features that addressed their requirements. For example, ETESTER lets product managers perform competitive analysis with competitors' apps and lets QA managers and DevOps perform continuous battery drain testing so they can maintain the status quo on the battery drain of their software.

Academic perfectionism vs. industry pragmatism. Our go-to approach as academics was to come up with principled solutions that could address the industry requirements. But in the industry, turnaround times often hold much greater value than the technical sophistication of the solutions. For example, we struggled quite a bit in achieving perfect repeatability in battery drain tests. Ultimately, since we could not control all sources of battery drain variations, we developed a statistics-based alert generation algorithm. We could have delivered this solution much sooner, albeit requiring a much greater number of test runs, without dwelling on trying to control the last remaining causes of non-repeatability.

Identifying serious users. From our industry interactions, we learned the hard way that engineers enjoy critiquing products and providing critical feedbacks even when they have no desire to use the product. For ETESTER, these feedbacks were of the form "We could use it only if ETESTER behaved like the product X or could seamlessly integrate with Y". Such feedbacks are dangerous because the feedback requirements can be hard to fulfill and even when they are fulfilled, they provide no real value causing wasted effort. Re-approaching such engineers with the fulfilled requirements almost always generates new requirements and excuses for not using the product.

On the other hand, engineers who were motivated to use ETESTER worked more closely with us, brainstorming together on simplifying their requirements to help us build ETESTER in incremental stages. We learned that it is important to identify seriousness of users early on by uncovering users' urgency and motivation. This can be done by asking business impact questions, instead of asking technical questions, like "How many engineers are currently devoted to working on battery performance?", "What is your deadline for having a battery testing setup?" and "How much budget have you assigned for a perfectly working battery testing setup?".

12 CONCLUSION

This paper reports our experiences over a span of 6 years in transforming an academic prototype of a "textbook" battery drain profiler into a fully automated battery drain testing and diagnostic tool usable by the mobile industry. We showed how ETESTER design and a novel testing methodology fulfilled 7 out of the 8 industry requirements. We further showed how the redesigned ETESTER was used to effortlessly find battery bugs in some of the most popular Android apps, such as Netflix and CNN. Our experience taught us many valuable lessons which we hope are helpful to systems researchers who care for real-world adoption of their research prototypes.

ACKNOWLEDGEMENT

We thank the anonymous shepherd and reviewers for their helpful comments. We thank all the software developers at Mobile Enerlytics who contributed to the development of ETESTER, which was supported in part by NSF grants SBIR Phase I 1549214 and Phase II 1660221.

REFERENCES

[1] Monsoon power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.
[2] Wonwoo Jung Authors Seokjun Lee, Yohan Chon, and Hojung Cha. Entrack: a system facility for analyzing energy consumption of android system services. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2015.
[3] Ning Ding and Y Charlie Hu. Gfxdoctor: A holistic graphics energy profiler for mobile devices. In *Proc. of ACM EuroSys*, 2017.
[4] Ning Ding, Daniel Wagner, Xiaomeng Chen, Abhinav̆ Pathak, Y. Charlie Hu, and Andrew Rice. Characterizing and modeling the impact of wireless signal strength on smartphone battery drain. In *Proc. of ACM SIGMETRICS*, 2013.
[5] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proc. of ACM Mobisys*, 2011.
[6] Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. Power modeling of graphical user interfaces on oled displays. In *Proc. of the 46th Annual Design Automation Conference*. ACM, 2009.
[7] Ehsan Ebrahimi. What are the acceptable values for the percentage deviation and the coefficient of variance?, 09 2018. https://www.researchgate.net/post/What_are_the_acceptable_values_for_the_percentage_deviation_DEV_and_the_coefficient_of_variance_CV.
[8] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proc. of WMCSA*, 1999.
[9] Jason Flinn and Mahadev Satyanarayanan. Energy-aware adaptation for mobile applications. *ACM SIGOPS Operating Systems Review*, 33(5):48–63, 1999.
[10] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proc. of ACM PLDI*, 1982.
[11] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4G lte networks. In *Proc. of ACM Mobisys*, 2012.

[12] Huawei kirin soc with npu. https://consumer.huawei.com/en/campaign/kirin980, Accessed: July 11, 2021.
[13] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. Ai benchmark: All about deep learn- ing on smartphones in 2019. In *Proc. of IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, 2021.
[14] Abhilash Jindal and Y Charlie Hu. Differential energy profiling: energy optimization via diffing similar apps. In *Proc. of USENIX OSDI*, 2018.
[15] Abhilash Jindal, Prahlad Joshi, Y. Charlie Hu, and Samuel Midkif. Unsafe time handling in smartphones. In *Proc. of USENIX ATC*, June 2016.
[16] Abhilash Jindal, Abhinav Pathak, Y. Charlie Hu, and Sam Midkiff. On death, taxes, and sleep disorder bugs in smartphones. In *Proc. of USENIX HotPower*, 2013.
[17] Dongwon Kim, Wonwoo Jung, and Hojung Cha. Runtime power estimation of mobile amoled displays. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013.
[18] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proc. of the ACM FSE*, 2014.
[19] Jiayi Meng, Qiang Xu, and Y. Charlie Hu. Proactive energy-aware adaptive video streaming on mobile devices. In *Proc. of USENIX ATC*, 2021.
[20] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proc. of ACM MobiCom*, 2012.
[21] Arvind Narayanan*, Xumiao Zhang*, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Denis Rybkin, Dustin Zhang, Michael Yang, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. A variegated look at 5g in the wild: Performance, power, and qoe implications. In *Proc. of the ACM Sigcomm*, 2021.
[22] Adam J Oliner, Anand P Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proc. of the ACM SenSys*, 2013.
[23] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging for smartphones: A first look at energy bugs in mobile devices. In *Proc. of Hotnets*, 2011.
[24] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proc. of EuroSys*, 2012.
[25] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system-call tracing. In *Proc. of EuroSys*, 2011.
[26] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of ACM MobiSys*, 2012.
[27] Feng Qian, Zhaoguang Wang, Alex Gerber, Z. Morley Mao, Sūbhabrata Sen, and Oliver Spatscheck. Characterizing radio resource allocation for 3g networks. In *Proc. of IMC*, 2010.
[28] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.
[29] Arjun Roy, Stephen M Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the Cinder operating system. In *Proc. of Eurosys*, 2011.
[30] A. Shye, B. Scholbrock, and G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *MICRO*, 2009.
[31] Get unstuck, learn better. https://socratic.org/.
[32] Google Edge TPU. https://cloud.google.com/edge-tpu?hl=en, Accessed: July 11, 2021.
[33] Matteo Varvello, Kleomenis Katevas, Mihai Plesa, Hamed Haddadi, and Benjamin Livshits. BatteryLab, a distributed power monitoring platform for mobile devices. In *Proc. of ACM HotNets*, 2019.
[34] B. L. Welch. The Generalization of Student's Problem When Several Different Population Variances are Involved. *Biometrika*, 34(1-2):28–35, 01 1947.
[35] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 479–494, New York, NY, USA, 2020. Association for Computing Machinery.
[36] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Proc. of USENIX ATC*, 2012.
[37] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proc. of ASPLOS*, 2002.
[38] Heng Zeng, Carla Schlatter Ellis, Alvin R Lebeck, and Amin Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *Proc. of USENIX ATC*, 2003.
[39] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R.P. Dick, Z.M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of CODES+ISSS*, 2010.
[40] Android Battery Historian. https://developer.android.com/topic/performance/power/battery-historian.
[41] Android Doze. https://source.android.com/devices/tech/power/platform_mgmt#doze.
[42] Android Energy Profiler. https://developer.android.com/studio/profile/energy-profiler.
[43] Android espresso. https://developer.android.com/training/testing/espresso.
[44] Android JobScheduler API. https://developer.android.com/reference/android/app/job/JobScheduler.
[45] Android Lint checks. http://tools.android.com/tips/lint-checks.
[46] Android Logging system overview. https://developer.android.com/studio/command-line/logcat#Overview.
[47] Android Perfetto. https://ui.perfetto.dev/.
[48] Android Performance Patterns video series. https://www.youtube.com/playlist?list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE.
[49] Profile Android app performance. https://developer.android.com/studio/profile/.
[50] Android Traceview. https://developer.android.com/studio/profile/traceview.html.
[51] Android Vitals. https://developer.android.com/topic/performance/vitals.
[52] AT&T Video Optimizer. https://developer.att.com/video-optimizer.
[53] AT&T Video Optimizer power model configuration file. https://github.com/attdevsupport/VideoOptimzer/blob/master/ARO.Core/src/main/resources/i997.conf.
[54] Brave browser. https://brave.com.
[55] Facebook Battery Metrics. https://github.com/facebookincubator/Battery-Metrics.
[56] Firebase test lab. https://firebase.google.com/docs/test-lab/android/overview.
[57] Headspin: Mobile & browser testing. https://www.headspin.io/platform/end-to-end-testing/.
[58] Apple MetricKit framework. https://developer.apple.com/documentation/metrickit.
[59] WWDC 2019: Improving battery life and performance. https://developer.apple.com/videos/play/wwdc2019/417/.
[60] Nimbledroid: Functional performance testing for Android & iOS. https://nimbledroid.com/.
[61] Qualcomm snapdragon profiler. https://developer.qualcomm.com/software/snapdragon-profiler.
[62] Robotium: User scenario testing for android. https://github.com/RobotiumTech/robotium.
[63] Systemtap. http://sourceware.org/systemtap/.
[64] Thermal mitigation. https://source.android.com/devices/architecture/hidl/thermal-mitigation.
[65] Measure energy impact with Xcode and Instruments. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/MonitorEnergyWithInstruments.html.