Combatting Energy Issues for Mobile Applications

Li, Xueliang; Chen, Junyang; Liu, Yepang; Wu, Kaishun; Gallagher, John Patrick

Published in:
ACM Transactions on Software Engineering and Methodology

DOI:
10.1145/3527851

Publication date:
2023

Document Version
Peer reviewed version

Citation for published version (APA):
Li, X., Chen, J., Liu, Y., Wu, K., & Gallagher, J. P. (2023). Combatting Energy Issues for Mobile Applications. *ACM Transactions on Software Engineering and Methodology*, *32*(1), Article 3527851. https://doi.org/10.1145/3527851

Combatting Energy Issues for Mobile Applications

XUELIANG LI, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
JUNYANG CHEN*, College of Computer Science and Software Engineering, Shenzhen University, China
YEPANG LIU, Research Institute of Trustworthy Autonomous Systems, Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, and Department of Computer Science and Engineering, Southern University of Science and Technology, China
KAISHUN WU, College of Computer Science and Software Engineering, Shenzhen University, China
JOHN P. GALLAGHER, Department of People and Technology, Roskilde University, Denmark and IMDEA Software Institute, Spain

Energy efficiency is an important criterion to judge the quality of mobile apps, but one third of our arbitrarily sampled apps suffer from energy issues that can quickly drain battery power. To understand these issues, we conduct an empirical study on 36 well-maintained apps such as Chrome and Firefox, whose issue tracking systems are publicly accessible. Our study involves issue causes, manifestation, fixing efforts, detection techniques, reasons of no-fixes and debugging techniques. Inspired by the empirical study, we propose a novel testing framework for detecting energy issues in real-world mobile apps. Our framework examines apps with well-designed input sequences and runtime context. We develop leading edge technologies, e.g. pre-designing input sequences with potential energy overuse and tuning tests on-the-fly, to achieve high efficacy in detecting energy issues. A large-scale evaluation shows that 90.4% of the detected issues in our experiments were previously unknown to developers. On average, these issues can double the energy consumption of the test cases where the issues were detected. And our test achieves a low number of false positives. Finally, we show how our test reports can help developers fix the issues.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Mobile Applications, Energy Issues, Energy Bugs, Android

*corresponding author

This work is supported in part by National Natural Science Foundation of China under Grant (61902249, 61932021, 62102265), in part by Open Research Fund from Guangdong Laboratory of Artificial Intelligence and Digital Economy (SZ), under Grant GML-KF-22-29, and in part by Guandong Basic and Applied Basic Research Fundation (Grant no. 2021A1515011562) and Guangdong Provincial Key Laboratory (Grant No. 2020B121201001).
Authors' addresses: Xueliang Li, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China, lixueliang01@gmail.com; Junyang Chen, College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China, junyangchen@szu.edu.cn; Yepang Liu, Research Institute of Trustworthy Autonomous Systems, Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China, liuyp1@sustech.edu.cn; Kaishun Wu, College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China, wu@szu.edu.cn; John P. Gallagher, Department of People and Technology, Roskilde University, Roskilde, Denmark and IMDEA Software Institute, Madrid, Spain, jpg@ruc.dk.

1049-331X/2022/1-ART1 $15.00
https://doi.org/10.1145/3527851

1 INTRODUCTION

Mobile devices have evolved into a wide ecosystem, providing millions of third-party apps to serve various user needs. Energy efficiency is a desirable quality attribute of mobile apps. However, many real-world mobile apps suffer from energy misuses. For example, Huang et al. [57] reported that most energy issues in mobile devices are caused by apps (47.9%) rather than systems (22.2%). We also arbitrarily sampled 98 open-source Android apps and found that 27 (30.3%) of them suffer from serious software energy issues.

Despite many pieces of existing work (e.g., [17, 54, 57, 64]), the characteristics and combatting-technology of energy issues are still not very well studied. Due to this reason, there exist few effective techniques to address serious energy issues. This motivates us to conduct an empirical study on 36 energy-inefficient Android apps. Specifically, we follow a classic empirical study's guideline [68] and design two sets of research questions. The first set attempts to study the characteristics of the energy issues:

- **RQ1 (Issue Causes):** *What are the common root causes of energy issues?*
- **RQ2 (Issue Manifestation):** *How do energy issues manifest themselves in practice?*
- **RQ3 (Issue Fixing Efforts):** *Are energy issues more difficult to fix than non-energy issues?*

The second set attempts to study the techniques in industry for detecting, diagnosing, and fixing energy issues:

- **RQ4 (Detection Techniques):** *What information and tools are the most helpful for detecting energy issues?*
- **RQ5 (Reasons of No Fixes):** *How many reported energy issues were left unfixed? Why could these energy issues not be fixed?*
- **RQ6 (Debugging and Fixing Techniques):** *What information and tools are the most helpful for debugging and fixing energy issues?*

We analysed 200 energy issues from the 36 open-source projects. The findings will motivate multiple lines of future research on combatting energy issues. Specifically, the following two findings inspired us to design a novel issue-detection technology. For RQ1, we identified four main root causes of energy issues: 1) *unnecessary workload*, 2) *excessively frequent operations*, 3) *wasted background processing*, and 4) *no-sleep* (**Finding 1**). For RQ2, we found that many energy issues require special inputs (64.6%) or special context (22.2%) to trigger and only 18.2% energy issues can manifest themselves with simple inputs (**Finding 2**).

We further studied the state-of-the-art testing technique for detecting energy issues [17], which was proposed by Banerjee et al., and made two observations. First, as shown in Table 1, the existing technique is only capable of exposing issues resulting from wasted background processing and no sleep. This is because the technique diagnoses energy issues based on *E/U ratio*, i.e., the ratio of energy-consumption to hardware-utilization. If E/U ratio is high, it means that energy consumption is high, while hardware utilization is low, implying that the app under analysis is energy-inefficient. This point of view seems reasonable. For energy issues caused by wasted background processing and no-sleep, the E/U ratio could be remarkably high. However, according to Finding 1, many energy issues can also be caused by unnecessary workload and excessively frequent operations. These issues cannot be detected via analyzing E/U ratio: they may enlarge E and U simultaneously, hence E/U ratio is not a good indicator of the existence of such issues. Second, regarding Finding 2, the existing technique is capable of generating simple and special inputs to trigger energy issues, but does not simulate special context (e.g., poor network performance). Due to this limitation, it may miss many real energy issues (22.2% of our studied issues can only be triggered under special context).

Based on these observations, we propose a novel testing framework for effectively detecting energy issues. We also performed experiments to evaluate our framework. The results show that our testing framework can uncover a large number of serious energy issues in high-quality apps, 90.4% of which have never been discovered before. On average, these issues double the energy cost of the corresponding test cases. Manual verification also

Table 1 Comparison with the existing technology.

Root Cause		Work [17]	This work
Unnecessary Workload	(42.7%)	✗	✓
Excessively Frequent Operations	(18.2%)	✗	✓
Wasted Background Processing	(22.7%)	✓	✓
No Sleep	(33.6%)	✓	✓
Manifestation Type		**Work [17]**	**This work**
Simple Inputs	(18.2%)	✓	✓
Special Inputs	(64.6%)	✓	✓
Special Context	(22.2%)	✗	✓

shows that our framework only reports a low number of false positives. Finally, we demonstrate how our test reports can facilitate developers in fixing the issues. The key contributions of this paper are as followed:

- To the best of our knowledge, we conducted the most comprehensive empirical study on developer-reported energy issues in mobile apps. Our study involves issue causes, manifestation, fixing efforts, detection techniques, reasons of no-fixes and debugging techniques. In contrast, the previous most relevant work [54] only investigated issue cause and fixing efforts.
- Inspired by the findings, we designed and implemented an automated testing framework for detecting energy issues. Our innovations include extracting battery-hungry input sequences from source code, steering the test direction on-the-fly for high detection efficacy, and employing machine learning to classify app workload.
- We empirically evaluated our framework and the results are promising: it detected 83 issues in 89 apps, creating many opportunities for optimizing the energy efficiency of these apps. As far as we know, this evaluation of a testing framework for energy issue detection is of the largest scale (*the largest previous evaluation [17]: 30 app subjects and detected 12 issues; this paper: 89 app subjects and detected 83 issues*). Our subjects are also of higher quality than previous work, which are selected considering metrics such as high popularity and maintenance quality. In constrast, most subjects in previous work do not meet this standard.

And this paper is an extended work of [48]. The key changes are as below:

- In [48], we only presented two aspects (i.e. issue causes and issue manifestation) of our empirical study. In this paper, we explore four more aspects to make more comprehensive and deeper understanding of energy issues. These four new aspects respectively are issue fixing efforts, detection techniques, reasons of no fixes, and debugging & fixing techniques. We will discuss about them individually in Section 3.3, 3.4, 3.5 and 3.6. These new findings will motivate multiple lines of future research on combatting energy issues.
- In Section 5.3.2, we explain in more detail about the determination of ϵ and MinPts. We present the key algorithm for determining them. We also employ communication apps as examples to explain the process.
- In Section 5.4, we discuss more specifically about the manual verification for the issue-candidates.
- In Section 6.2, we show how the parameters in our testing framework influence the results of false positives and false negatives.
- In Section 6.2 again, we conduct new experiments on a new device and new apps to evaluate the generality of our testing framework. Meanwhile, we also experimentally justify the effectiveness of our steering algorithm.
- We add a new section (Section 6.4) to show how our test reports can assist in fixing energy issues. We use two concrete examples to illustrate the debugging procedures.

- In Section 6.4, we operate a user study on developers to evaluate the usefulness and usability of our testing framework.
- In the beginning of Section 9 (related work), we add discussion about the relation between performance issues and energy issues.
- In Section 3.3, we add the statistics of the issues from all app subjects, not limited to Chrome and Firefox.
- The dataset of issue reports for our empirical study is updated to June 2021, which guarantees the timeliness of our study

In the remainder of this paper, we first introduce the data source for empirical study in Section 2, and discuss the findings in Section 3. We present our testing framework and technical details in Sections 4 and 5. Finally, we present an evaluation of our framework and discuss the results in Section 6.

2 DATA SOURCE

Open-source projects typically have publicly accessible issue tracking systems and code repositories. In the issue tracking systems, developers can post an issue report, which contains a title and a main body part, to report the symptoms of their observed bug/issue [1] and the steps to reproduce the issue (optional). Following that, developers can discuss the issue and comment on the report. Those developers who are assigned to fix the issue can propose potential code revisions. Typically, after code review by other project members and further changes, such revisions will be committed to the project's code repository.

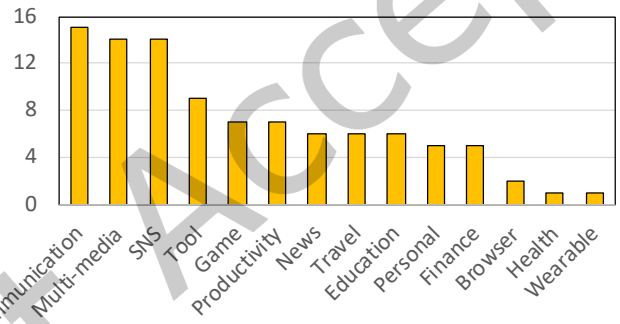


Fig. 1 The 98 app subjects of different categories.

Our empirical study is conducted on well-maintained Android application projects from three popular open-source software hosting platforms: GitHub[2], Mozilla[3], and Chromium[4] repositories. The criteria for selecting app subjects for our study are these: 1) a subject should have achieved at least 1,000 downloads on the market (popularity), 2) it should have more than one hundred code revisions (maintainability), 3) its issue tracking system and code repository should document the details of various issues such as what tools and information utilized to diagnose the issues (informativeness). Following these criteria, we arbitrarily selected **98** app subjects from those three software hosting platforms. These open-source applications are also indexed by the F-Droid database[5].

[1] We use the terms of bugs and issues interchangeably in this paper.
[2] github.com
[3] dxr.mozilla.org bugzilla.mozilla.org/home
[4] www.chromium.org bugs.chromium.org/p/chromium/issues/list
[5] https://f-droid.org/

Figure 1 presents the numbers of app subjects of different categories. They cover most application categories in F-Droid and in total contain 2,935,090 issue reports, indicating that these subjects are quite large-scale.

To search for energy issues, we adopted a semi-automated approach as described below.

Explicitly-labeled energy issues. First, we studied the issue labels used in each project's issue tracking systems. Only two repos, Chrome and Firefox, have explicit labelled energy issues. In Chrome, 57 issues are labeled with "*Performance-Power*" or "*Performance-Battery*". In Firefox, only three issues are labeled with "*power*". We first include these 60 issues in our dataset.

Energy issues without explicit labels. For other repos, where energy issues are not explicitly labeled (different developer teams have different practices), we employed keyword searching in the issue reports' title and body to locate potential energy issues. Since developers may not carefully label each issue report, we also applied keyword searching in Firefox's and Chrome's issue reports. The Cartesian product rules listed in Formula (1) and (2) describe the keywords we used. According to the rules, for instance, "*energy consumption*", "*power save*" and "*battery hit*" can be generated as the search keywords.

$$\{energy,\ power\} \times \{consumption,\ save, efficency, use,\ usage,\ draw\} \tag{1}$$

$$\{battery\} \times \{drain,\ drop,\ draw,\ use,\ usage,\ save,\ conserve,\ hit\} \tag{2}$$

While the general keyword searching can help retrieve most energy-related issue reports, it can also produce false positive results when the issue reports accidentally contain any of our keywords. For example, Dash Clock issue 593 reports that the device can not be charged when using the app[6]. The report body *"using this app the battery does not charge"* contains keywords *"battery"* and *"use"*. However, this report is not reporting an energy issue in mobile apps. To filter out such irrelevant issue reports, after keyword searching, we manually verified each returned issue report to make sure the issue concerned is indeed an energy issue. Note that, there also exist duplicate issues in the issue tracking systems, and the issues are explicitly labelled by the developers. We thus remove the duplicate issues based on such labels. In total, we checked 1352 retrieved issue reports and this helped us locate 200 real energy issue reports from **36** apps.

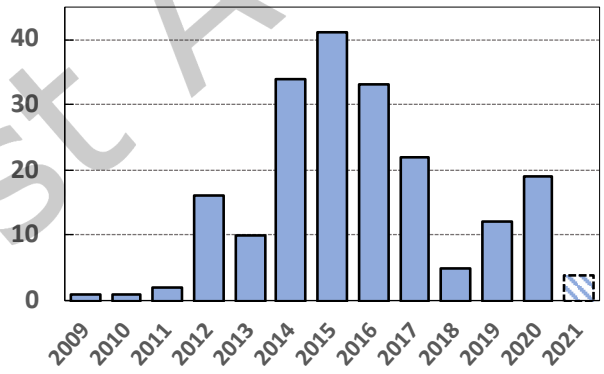


Fig. 2 The yearly number of reported energy issues.

Figure 2 shows the distribution of opening dates of these reports. There was a peak reporting period from 2014 to 2017. Only 26.5% (53 out of 200) reported issues were fixed. The main reasons for no fixes are these: 1) many energy issues are irreproducible (55.7%), 2) the problematic code cannot be localized (7.8%), 3) energy-saving by

[6]"Dash Clock" is the app name, "593" is the issue's ID given by the corresponding issue tracking system.

fixing the issues cannot be evaluated (7.8%), and 4) the fix causes other issues (7.8%). So in this paper, we will present a cutting-edge technology for effectively pinpointing energy issues in the lab where testing condition is precisely controlled, so the issues can be strictly reproduced and localized. Also, energy-saving can be accurately evaluated.

3 EMPIRICAL STUDY

To answer our research questions, we carefully studied the 200 energy issue reports.

In empirical studies, usually, for classification-like RQs (e.g. types and proportions of issue causes), we do not explicitly propose hypothesis since there may exist infinite possibilities. Choosing any certain case as hypothesis will probably result in a denial.

So no explicit hypothesis for classification-like RQs is how the literature practises, as shown in [54, 56]. The classification and quantification on their own are very informative and helpful for developers and researchers, and are the reflections to the RQs and the learned insights. Such style of empirical studies like [54, 56] already proved its significant value for the industry and the academia.

But for if-or-not RQs (e.g. "RQ3: Are energy issues more difficult to fix than non-energy issues?"), proposing hypothesis is a standard approach for empirical study. And we will apply this standard for the RQ3.

3.1 **RQ1:** What are the common root causes of energy issues?

Among the 200 reports, 110 explicitly show the information on root causes of the issues. We examined all of them and observed the following six root causes. Some issues were caused by multiple reasons, hence, the sum of the percentages below is over 100%.

Unnecessary workload (47/110=42.7%). Many applications perform certain computations that do not deliver perceptible benefits to users. These computations incur unnecessary workload on hardware components including CPU, GPU, GPS, network interface, and screen display. For example, in the reports of Chrome issue 662012 and 541612, the application produces frames constantly even when visually nothing is changed or repainted, which causes huge workload on CPU and GPU. And the report of OpenGPSTracker issue 406 shows that the app keeps recording users' location even after not moving for minutes, barely exhausting battery.

Excessively frequent operations (20/110=18.2%). Performing certain operations too frequently can also waste power. In comparison with unnecessary workload, when fixing energy inefficiencies caused by excessively frequent operations, the developers do not completely remove the operations (because their functionality is necessary), but reduce the frequency of operations. For example, in Firefox (issue 979121), whenever users type in the URL bar and the text changes, the app will query the database (e.g. for auto-completion). Considering users often visit the websites they visited before, developers suggested to store users' browsing history in memory to reduce database hits to save energy. Another example is Firefox issue 1057247, where developers think that the frequency for retrying to fetch failed favicons is too high. Their patch to fix the issue lowered the frequency, as shown in Program 1.

Program 1 Part of Java patch for Firefox issue 1057247.

```
- // Retry failed favicons after 20 minutes.
- public static final long FAILURE_RETRY_MILLISECONDS = 1000 * 60 * 20;
+ // Retry failed favicons after one hour.
+ public static final long FAILURE_RETRY_MILLISECONDS = 1000 * 60 * 60;
```

Wasted background processing (25/110=22.7%). As battery powered mobile devices are extremely sensitive to energy dissipation, it is a good practice to make backgrounded applications as quiet as possible. Specifically, the

"backgound" here means that after the use of an application or "activity" (a major type of application component that represents a single screen with a user interface[7]), users press the *Home* button or switch to another application or activity, so the previous application or activity goes to background. Typical examples are Chrome issue 781686 and Firefox issue 1022569: when users select a new tab (each tab is an activity), the invisible old tab would still keep being reloaded, which wastes battery. But these issues do not have an easy solution since users may want old tabs to be reloaded.

Program 2 JavaScript patch of Firefox issue 1026669.

```
case ''ssdp-service-found'':
- {
-  this.serviceAdded(SimpleServiceDiscovery.findServiceForID(aData));
-  break;
- }
+ this.serviceAdded(SimpleServiceDiscovery.findServiceForID(aData));
+ break;
case ''ssdp-service-lost'':
- {
-   this.serviceLost(SimpleServiceDiscovery.findServiceForID(aData));
-   break;
- }
+ this.serviceLost(SimpleServiceDiscovery.findServiceForID(aData));
+ break;
+ case ''application-background'':
+ // Turn off polling while in the background
+ this._interval = SimpleServiceDiscovery.search(0);
+ SimpleServiceDiscovery.stopSearch();
+ break;
+ case ''application-foreground'':
+ // Turn polling on when app comes back to foreground
+ SimpleServiceDiscovery.search(this._interval);
+ break;
```


No-sleep (37/110=33.6%). The no-sleep issue means that when the screen is off and device is supposed to enter sleep mode, certain apps still keep the device awake, which usually results from misuse of asynchronous mechanisms [59] like services, broadcast receivers, alarms and wake-locks. For example, Kontalk issue 143 unnecessarily holds a wake-lock, preventing the device from sleep. For another example, in Firefox (issue 1026669), the Simple Service Discovery Protocol (SSDP) activates the searching service every two minutes when the screen is off, which not only incurs a large amount of workload but also prevents the device from entering the sleep mode. Program 2 gives the JavaScript patch for fixing this issue. It added the cases to deal with "application-background" and "application-foreground" for the SSDP service. Note that, the "application-background" defined by developers includes both screen-off time and the scenarios where users switch to another application. So this issue belongs to two categories: *no-sleep* and *wasted background processing*.

Runtime exception (3/110=2.7%). In some cases, runtime exceptions may provoke abnormal behaviors of a mobile application and cause energy waste. For instance, in AntennaPod (issue 1796), the "NullPointerException" makes the download process persist and consume power. In our study, such energy issues caused by runtime exceptions are not common and we only observed three cases.

Spike workload (2/110=1.8%). A spike workload can cause lagging UI [56], degrade user experience and heat up the device, inducing a huge energy waste. For instance, in RocketChat (issue 3321), when users send or receive GIF animation pictures, CPU utilization quickly rises to 100% and heavily affects the battery.

[7] https://developer.android.com/guide/components/fundamentals.html

The above analysis proves that unnecessary (the first, second and fifth root causes) or unoptimized (the third, fourth and sixth root causes) workloads are the major causes of energy issues. This observation enables us to give a better definition of energy issues:

Definition 1: *energy issues are the issues that incur unnecessary or unoptimized energy use, and noticeably harm user experience.*

3.2 **RQ2:** How do energy issues manifest themselves in practice?

Out of the 200 reports, 99 contain explicit information that shows how the issues manifest themselves. We studied these 99 issues to answer RQ2. We observed three manifestation types: *simple inputs*, *special inputs* and *special context*. It is worthwhile to notice that, *simple inputs* and *special context* may combine to incur issues, on the other hand, *special inputs* and *special context* may also overlap.

Simple inputs (18/99=18.2%). Simple inputs mean one tap or swipe gesture in common interaction scenarios. We found 18 issues are of this type of manifestation. For example, Andlytics issue 543 lets the app refresh itself whenever the user opens the app. And VoiceAudioBookPlayer issue 299 makes the app unnecessarily scan folders every time the user starts or leaves the app.

Special inputs (64/99=64.6%). The majority of the energy issues can only be triggered with certain specific inputs or a sequence of user interactions (e.g. text typing, taps, or swipes) under certain states of an application. For instance, the c:geo (a geocaching app) issue 4704 requires three steps to reproduce: 1) open the app and make sure GPS is inactive since the app starts, 2) change between cache details and other tabs of the same geocache, so GPS is activated, 3) put the device in standby and let timeout to screen-off. After a while, users would find GPS stays active even when the screen is turned off. To avoid energy waste, users decide to quit using the app.

Special Context (22/99=22.2%). Special context includes environmental conditions (rather than user interactions, e.g. taps) such as the accessibility of networks, location of the device, settings of the OS and applications. In our dataset, 22 issues require such special context to trigger. For instance, MPDroid issue 3 appears when the user is watching stream videos but the network is disconnected; the app then keeps trying to load the video and consumes battery. AnkiDroid issue 2768 occurs when users lock the phone screen when the application is in "review" mode and a notification comes in afterwards, so the screen will hold on until the battery is dead.

3.3 **RQ3:** Are energy issues more difficult to fix than non-energy issues?

For this research question, we propose the hypothesis as below:

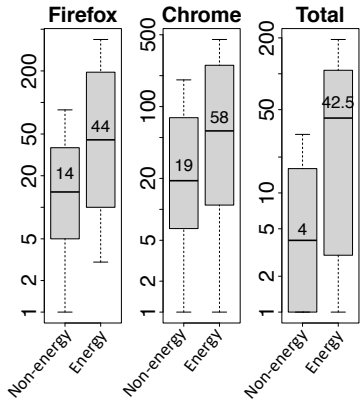
H: Energy issues are less or equally difficult compared with non-energy issues.

We raise such hypothesis because so far we have seen no evidence that energy issues would require more efforts to fix. In this section, we will statistically compare the effort for fixing energy issues with that for non-energy issues.

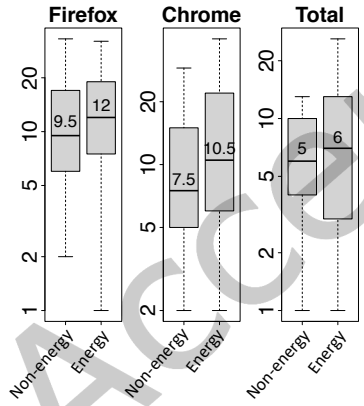
Table 2 Bug fixing effort

Criterion	Min	Median	Max	Mean
Open duration (#days)	1	42.5	1054	117.5
#Comments	0	6	144	12.0
Patch size (#changed lines)	2	88	3033	237.7

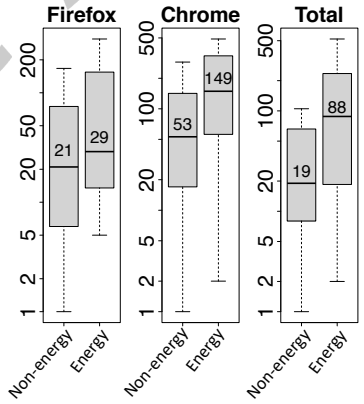
We use three metrics to assess the fixing effort of energy issues: (1) *issue open duration* (i.e. for how long the correponding issue reports are open), (2) *the number of comments* (indicating how much developers discuss when diagnosing the issues), and (3) *the patch size* (i.e. the lines of code developers write to fix the issues).



(a) Open duration (# days).



(b) Number of comments.



(c) Patch size (# changed lines).

Fig. 3 Bug fixing efforts.

Table 2 gives the min, median, max, and mean of these metric values. Table 2 is based on the 53 fixed energy issues (see soon in Section 3.5). And 40 out of these 53 issues are from Firefox and Chrome. Even though the number of fixed energy issues seems not large, these issues are all the fixed energy issues in the whole data source of 2,935,090 issues. It is because energy issues are difficult to be detected and fixed.

On average, the open duration for energy issue reports is around four months, implying that after energy issues are detected and reported to developers, it still requires much time to diagnose and fix the issues. The average number of comments in energy issue reports is 12.0, which is not noticeably prominent. On the other hand, the average lines of code to fix energy issues is as large as 237.7, indicating a non-trivial amount of efforts.

We further compared the fixing efforts of energy issues with those of non-energy issues. We made the comparison on three energy-issue data sets: Firefox, Chrome and "Total". Firefox and Chrome represent strong and experienced development teams with well-maintained issue-tracking systems. "Total" stands for an overall development experience of developers. Specifically, 10% of the energy issues in Total are from Firefox and Chrome, the other (90%) are from relatively small app projects held on GitHub.

And for the comparison, we randomly selected 100 non-energy issues from each data set. The used three metrics are the same as above. Figures 3(a)–3(c) presents the comparison results using box plots. As we can see, on average, the open duration of energy issue reports is three to ten times longer than that of non-energy issues. The numbers of comments for both types of issues are comparable at a similar level. The patches to fix energy issues are much larger than those for fixing non-energy issues. From these observations, we can conclude that fixing energy issues is generally more challenging than fixing non-energy issues. So the hypothesis H is denied.

And in the following we will explore and discuss about the reasons why energy issues are difficult to fix and what techniques can help developers diagnose and fix energy issues in practice.

3.4 **RQ4:** What information and tools are the most helpful for detecting energy issues?

When investigating this research question, we found that among the 200 issue reports we collected, 1) there are 188 issues with reports mentioning what information helped developers confirm the existence of the corresponding issues and 2) there are 83 issue reports mentioning the tools developers used for detecting the corresponding issues. Table 3 and Table 4 present these helpful information and tools, respectively.

Before discussing our findings, we first clarify some details in Table 4: *Percentage* (*Pct.*) means the information provided by the corresponding tool is presented to developers as a percentage and *Number* (*No.*) means the information is presented as a number. *Accumulative* (*Acm.*) means the information is the accumulation of the corresponding metric values, such as the total energy use for four hours. *Instantaneous* means the information is the instantaneous value of the corresponding metrics, such as the energy use at a certain moment (i.e., power).

As we can see from Table 3, there are nine types of information that are helpful with energy issue detection.

- In most cases (for 98 issues), developers confirmed the existence of energy issue by checking the measured or predicted **energy use** at several levels of granularity such as device, app, and process according to which tool is employed. `MBDA` can only provide device-level energy information. On the other hand, `Settings`, `ADB` and etc. can provide app-level information. Specially, the tool `Telemetry` used by Chrome developer can run benchmarks of a set of representative web pages and measures the energy consumption of rendering each page so that the energy uses of individual pages (scenarios) can be compared.
- There are 48 energy issues detected according to the developers' and users' **knowledge** or **engineering experience**. For example, in Firefox issue 736602, the developers suspect that the frequency for updating inactive tabs is too high and suggest reducing the frequency to save battery power. This type of energy issue detection requires the developers to be knowledgeable of the application's source code and the device's energy features.

Table 3 Helpful information for issue detection

Category	Granularity	Use Rate	
Energy use	device	40/188 = 21.3%	52.1%
	app, process component	49/188 = 26.1%	
	scenario	9/188 = 4.8%	
Engineering experience		48/188 = 25.5%	25.5%
Non-energy Info.		29/188 = 15.4%	15.4%
CPU use	device	2/188 = 1.1%	11.2%
	app, process	13/188 = 6.9%	
	method	6/188 = 3.2%	
Awake time	device	1/188 = 0.5%	4.8%
	app	5/188 = 2.7%	
	thread	3/188 = 1.6%	
Network use	app	4/188 = 2.1%	2.1%
Debugging print		3/188 = 1.6%	1.6%
GPS use	app	2/188 = 1.1%	1.1%
GPU use	method	1/188 = 0.5%	0.5%

- 29 energy issues also incur **non-energy information**, such as the app's abnormal behavior, when the corresponding application is interacting with users. For instance, Firefox issue 732723 freezes the application when users choose to print a web page into a PDF file. MPDroid issue 3 occurs when the network is reconnected after disconnection and the streaming is not correctly restarted, but the application keeps running and draining battery.
- For 28 issues, developers detected them by checking the usage information of individual hardware components including **CPU**, **GPU**, **GPS**, and **network**, which are good indicators of energy uses. For instance, c:geo issue 3009 was exposed by inspecting the GPS usage of the application using the tool `Settings`, which provides how long the application has been using GPS in the format of seconds. Another issue (Chrome issue 462752) was revealed by using the Linux `Top` tool, which shows the instantaneous CPU utilization of the application in the format of percentages.
- For detecting nine no-sleep issues, developers used tools such as `Settings`, `ADB` and `BBS`, to generate app-level **awake time** information, and `BMW` to check coarse-grained device-level awake time. Some developers also used the tool `WLD` for detecting no-sleep issues. The difference between `WLD` and the former four tools is the following. First, `WLD` can provide thread-level awake time information, the others can only produce app-level or device-level information. Second, the other four tools can provide the information on CPU usage, GPS status, network packages in addition to awake time information, but `WLD` only has awake time information.
- There are three issues detected by investigating the **debugging print**. For example, AntennaPod issue 1796 is noticed because the message "processing downloads" persists in the debugging print, and hence developers realized there are some unnecessary processing in the background.

3.5 **RQ5:** How many reported energy issues were left unfixed? Why these energy issues could not be fixed?

Among our collected 200 issues, only 53 (26.5%) are fixed, and 147 (73.5%) are unfixed. For 115 of the 147 unfixed issues, we figured out the major reasons why they are not fixed by inspecting the issue reports. Specifically, we leverage natural language understanding (NLU) to help automatically label the issue reports. We first manually

label each issue report with a certain category (e.g. "irreproducible"). Then we feed NLU model with the reports and the corresponding labels to train the model. Lastly, we use the model to label new issue reports. Finally, we conduct a brief manual check for all the automatically-labelled reports.

We discuss our findings in the following.

Irreproducible (64/115 = 55.7%**).** The most prevailing reason is that the developers can not reproduce the issue even though the issue appeared once and was noticed by developers. It is due to lack of information to duplicate the issue manifestation process. For instance, K-9 Mail issue 1290 can not be reproduced since no Android version is specified. And Firefox issue 732723 is irreproducible as a result of missing device model information.

Table 4 Helpful tools for issue detection and localization

Tools	Energy		CPU		GPS		Network		Awake	Trace	*UseRate*
	details	GL[7]	details	GL	details	GL	details	GL			
Settings[1]	Pct.[4]Acm.[5]	app	No.[16]Acm.	app	No.Acm.	app	No.Acm.	app	app	battery level	56.6%
Telemetry [9]	No.[6]Acm.	scenario									13.2%
Top[2]			Pct. Inst.[8]	app							8.4%
TV [10]			Pct.Acm.	**method**							4.8%
WLD [11]									thread		3.6%
BBS [3]							Pct.Acm.	app	app		3.6%
Proc[3]			No.Acm.	app							2.4%
ADB [1]	Pct.Acm.	app	No.Acm.	app	No.Acm.	app	No.Acm.	app	app	hardware usage	1.2%
BMW [2]	Pct.Acm.	app	No.Acm.	app	No.Acm.	device	No.Acm.	device	device	battery level	1.2%
GDB [4]										**method**	1.2%
ZDBox [12]	Pct.Acm.	app									1.2%
MBDA [7]	Pct.Inst.	device								drain rate	1.2%
Perfetto [8]			No.Acm.	app						app	1.2%

1. *Battery usage, Android Settings* 2. *Linux "top" Command* 3. *Linux "/proc/<PID>/stat" files*
4. *"Pct." means "Percentage"* 5. *"Acm."* means "Accumulative" 6. *"No." means "Number"*
7. *"GL" means "Granularity Level"* 8. *"Inst." means "Instantaneous"*

The above-mentioned information is static. There are many cases where reproducing an energy issue requires much more complicated contextual information. This is because mobile applications run in environments, which are far more dynamic and complex than application running on laptops and PCs. To reproduce issues in mobile applications, developers need information about hardware and OS activities, as well as information from external and dynamic sources, such as user inputs, signal strength and GPS status. There has been comprehensive study on this problem. Tools such as [14, 39, 49] can capture useful information from single source for issue reproducing. Furthermore, [13] proposes an approach which can efficiently obtain multiple-source information to facilitate developers to reproduce and diagnose the issues.

Unable to localize problematic code (9/115 = 7.8%**).** Even though some issues could be reproduced and the energy waste symptoms can be observed, there is still a challenge to pinpoint the root cause. In our dataset, we found nine issues that are detected and reproduced by developers, but were not fixed due to the reason that developers were unable to locate the problematic code. Take Firefox issue 1217871 as an example. It was detected with the information of device-level energy use (the battery drains from 89% to 50% in 1.5 hours); however, developers cannot find the exact code region that caused the energy waste. Later in Section 3.6, we will further analyze what information is helpful for localizing the issue root causes.

Unable to evaluate energy-saving (9/115 = 7.8%**).** As shown in nine issue reports, the developers did not fix the issue because they cannot assess whether the energy-saving is worthwhile for their efforts. In the report of Firefox issue 823582, developers cannot decide whether to store the data into files or databases because they are not aware which way is more energy-efficient. In another report of c:geo issue 3009, the developers did not trust the precision of energy profiles provided by `Settings` because such information is estimated purely relying on several simple system readings such as CPU and GPS utilization and screen-on time. On the other hand, they had no access to physical devices that can precisely evaluate whether their code changes would lead to energy saving. Therefore, in the end, they gave up the code optimization and left the issue unfixed.

Causing other issues (9/115 = 7.8%**).** Nine issues were left unfixed because developers' patches would cause "other issues" include regression in the application's performance, maintainability and etc. For instance, one patch to fix Firefox issue 751681 makes the startup time of the application much longer. The fix of c:geo issue 1557 is too complicated and results in reforming the structure of software, making the project harder to maintain. And Luis Cruz et al. [24] is dedicated to investigating this problem.

Negligible energy-saving (2/115 = 1.7%**).** In two issue reports, the developers mentioned that the energy saved by their code refactoring is too insignificant, and hence it is pointless to allocate effort. For example, in the report of Firefox issue 899424, one developer notices that 0.3% CPU time is spent on executing an uninteresting function, which results in energy waste. However, as the benefit of code optimization is negligible, other members of the project were not convinced to focus effort on it.

Others (25/115 = 21.7%**).** There are 25 issues that have not been fixed for other non-technical reasons. For example, some issues were reported not long ago and the diagnosis is still in process, as shown in Signal issue 11067. For another example, the project was already dead before Vanilla Music issue 121 was reported. There are also cases where developers decided to fix the issues in the future releases of their applications and thus suspended the debugging process (e.g. Signal issue 4437, 3641 are to be fixed in the next release).

3.6 **RQ6:** What information and tools are most helpful for debugging and fixing the issue?

Apart from the issues detected via developers' engineering experience and non-energy information, the rest are detected by tools as listed in Table 4. And Table 5 presents the fixing rates of energy issues when provided with different information from these tools. Generally, finer-grained information produces better fixing rates. For the energy and hardware-component usage information, the granularity of device-level and app-level can only result in a fixing rate below 25.0%. But method-level can effectively pinpoint the issues, and has helped developers

Table 5 Helpful information for fixing issues

Category	Level	#fixed[1]	#total[2]	Rate[3]
Energy use	device	5	40	12.5%
	app	7	45	15.6%
	scenario	0	9	0%
CPU use	device	0	2	0%
	app	3	12	25.0%
	method	5	6	83.3%
Network use	app	0	4	0%
GPS use	app	0	2	0%
Awake time	device	0	1	0%
	app	2	5	40.0%
	thread	2	3	66.7%
Debugging print		2	3	66.7%

1. $\#fixed$ is the number of issues **fixed** using this info
2. $\#total$ is the number of issues **detected** and **diagnosed** using this info
3. $Rate = \#fixed/\#total$

resolve 83.3% of the detected issues. For awake time information, thread-level is very supportive (66.7%) in fixing issues. It is worth mentioning that the tools that generate method-level profiles are `GDB` (for execution trace) and `TV` (for CPU use, now `TV` is deprecated, developers can use CPU Profiler[8] instead). The tool that provides thread-level awake time is `WLD`.

Table 5 also demonstrates the comparison between RQ4 and RQ6. In the table, #total is the number of issues detected and diagnosed using the tools that can provide the corresponding information (w.r.t. RQ4). #fixed is the number of issues fixed using the information (w.r.t. RQ6). The results in Table 5 indicate that even though many energy issues can be detected using certain tools, it is still a big challenge to fix them. More importantly, Table 5 also tells that more fine-grained information is more helpful for fixing issues. For example, method-level CPU information helps fix 83.3% of detected issues, but device-level helps no fix.

In addition, we can also observe that only three energy issues are diagnosed using traditional debugging information, i.e. debugging print, as shown in the last row of Table 5. This indicates the cumbersomeness of such traditional debugging information in detecting energy issues. But once the issues are captured by using this information, they can be fixed in most cases (66.7%).

4 OVERVIEW OF TESTING FRAMEWORK

The findings of our empirical study will benefit many aspects of research on addressing energy issues. And the following two findings motivated us to design a novel testing framework for effectively **detecting** energy issues in real apps:

- From **Finding 1**, we address previously unaddressed energy issues caused by unnecessary workload and excessively frequent operations.
- From **Finding 2**, we found that 24.2% of energy issues can only be manifested under special context such as poor network performance. However, such factors were neglected.

According to the first observation, as we discussed in Section 1, *evaluating the necessity of app workload is crucial for identifying these issues*. We will use machine learning to cluster workloads, and further assess their necessity, as shown later in Section 5.3. According to the second observation, we will devise two types of most

[8]https://developer.android.com/studio/profile/cpu-profiler

common special context for effectively revealing these issues, as shown in Section 5.1.2. Developers also can duplicate our experiment for detecting these hidden issues. Importantly, we also make practical designs and implementations to enhance the efficacy of our testing framework.

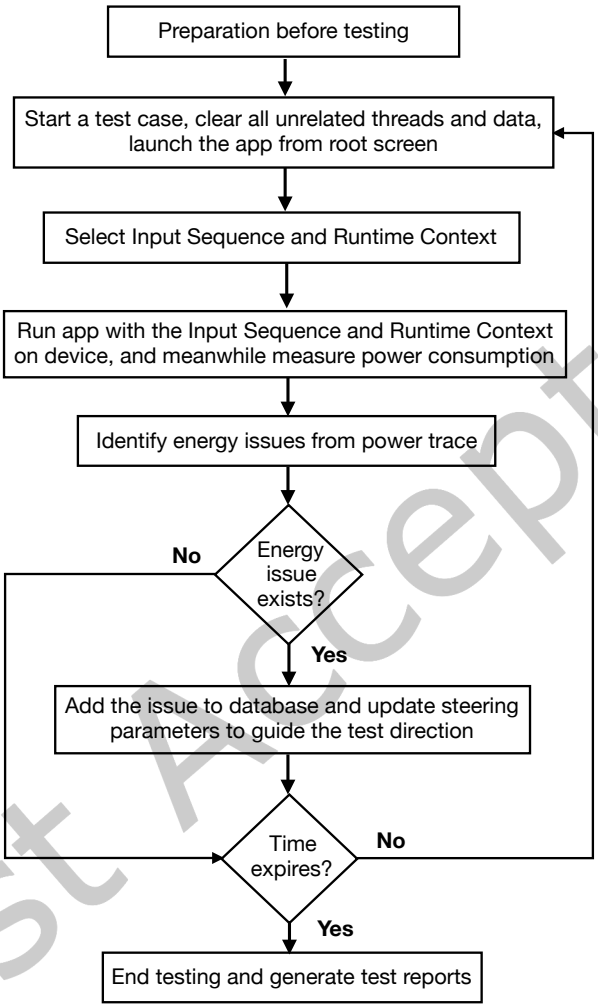


Fig. 4 The flow chart for our testing framework.

Figure 4 shows the framework overview. The framework first makes sophisticated preparation before testing. For example, it inspects the source code and collects the candidate input-sequences that are most suspected of energy overuse. A set of candidate runtime contexts (containing the above mentioned two types of special contexts) are also carefully designed to increase the chance of provoking energy issues. Later, these candidate inputs and contexts will be explored under an effective and systematic scheme.

To start a test case, the framework at first clears unrelated threads and data to minimize the interference from other applications and previous test cases. Then, it will select one input sequence and one runtime context from the candidates, then run the app with them. During the entire test, the power consumption of device is traced

with a power monitor. Our framework will look into the power trace and decide whether an energy issue exists. If one does exist, the issue information will be added into database. The entire test is limited with a time budget. If the time budget runs out, the test will quit and test reports will be generated for developers to help fix the issues. Otherwise, our framework will start a new test case.

As mentioned above, our framework explores the inputs and contexts under a systematic scheme, where the exploring direction is tuned on-the-fly. The rationale of our scheme is that if an energy issue occurs, it implies that the type of the input sequence and runtime context incurring this issue may be more likely to uncover energy issues than average case, since it did cause an energy issue to show up; we thus increase the chance of this type of inputs and context to be tested. Concretely, we utilize a set of parameters and iteratively update them to guide the test direction, as shown later in Section 5.2.

The large-scale evaluation (Section 6) shows that, exploiting these practical and targeted tests, our framework largely outperforms the state of the art on the efficacy in detecting all kinds of energy issues.

5 DETAILED TECHNOLOGY

This section introduces detailed implementations of our testing framework. It involves how to design candidate input sequences and runtime contexts, how to steer the test direction at runtime, and how to identify energy issues from the power traces, etc. The overall objective is to effectively and accurately pinpoint energy issues.

5.1 Preparation before testing

As shown in Section 4, our test case is driven by the input sequence and runtime context. Our candidate input sequences are designed for high utilization of the main hardware components, such as CPU, screen display and network interface, since they are usually the culprits of energy waste, as shown in the literature [75]. Meanwhile, according to Finding 2, we will carefully devise a set of artificial runtime contexts for effectively detecting those energy issues.

5.1.1 Design of candidate input sequences. We design two types of candidate input sequences. One is **weighted input sequences**, the other is **random input sequences**. The former helps our framework detect issues in a guided manner, the latter will cover more paths that are hard to predict in the apps.

Weighted input sequences are generated referring to the Event-Flow Graph (EFG) [60]. Each node in an EFG is a User Interface (UI) component, such as a button or a list item. If a user interaction on a UI component, say $node_1$, can immediately bring out another UI component, say $node_2$, then the EFG has a directed edge from $node_1$ to $node_2$. Technically, we utilize Layout Inspector[9] to construct the EFG of an app. *An arbitrary path in EFG could be a candidate input sequence for our testing framework.* In practice, our test cases always start from the root node (i.e. the initial UI component of the app). The lengths of paths are constrained with a limit. Note that, even though Layout Inspector can construct the EFG, it does not have a capacity to run apps with the paths. So in our test, we use Dynodroid[10] to feed the paths to apps. Detailed implementation is shown in [17]. Importantly, all input sequences generated from EFG are assigned a weight. The weight indicates potential of a sequence to cause energy waste; an input sequence with a larger weight has a higher priority to be tested. We use Equation (3) to calculate the weight for each input sequence. S is the number of a certain set of system APIs invoked by the input sequence. C is the number of function invocations and block transitions incurred by the input sequence. α and β are used for adjusting influences of S and C on the weight; $\alpha > 0, \beta > 0, \alpha + \beta = 1$. In practice, we set α at 0.6 and β at 0.4.

$$weight = \alpha * S + \beta * C \tag{3}$$

[9]https://developer.android.com/studio/debug/layout-inspector
[10]https://dynodroid.github.io

The reason why we resort to S and C to indicate the potential of excessive energy use is this: as shown in [75], the main energy-consuming components are CPU, screen display, network interface (cellular and WiFi), as well as GPS and various sensors. Except for CPU, all other components can only be controlled by a set of system APIs, as listed in [17]. The more this set of system APIs an input sequence accesses, the larger are the chances it causes energy waste. The CPU executes basic operations that constitute the source code of apps, for example, arithmetic operations like additions and multiplications, and control-flow operations mainly including function invocations and block transitions. Recent research [43, 47] has shown that control-flow operations are the actual main energy-consumers for Android app source code. We therefore use the total number of the main control-flow operations (i.e. function invocations and block transitions) to indicate the potential CPU overload of an input sequence.

Note again that, we calculate S and C before testing. We first instrument the app source code, and run the app with the input sequences in the emulator on a powerful PC, and then record their S and C values individually.

Apart from **weighted input sequences**, we also designed **random input sequences** to cover exceptional cases we might not envisage. We use the Monkey tool[11] to generate random input sequences, such as taps and swipes. "Random" here means the position of the inputs on screen are randomly set.

5.1.2 Design of candidate runtime contexts. The entire experiment is set in a signal shielding room, enabling us to manipulate the contextual factors, such as the strength of WiFi (in our test, we employ WiFi as the connection to Internet) and GPS signal. We designed three types of runtime contexts, namely, `Normal`, `Network Fail` and `Flight Mode`. In `Normal`, the WiFi and GPS work normally (package delivery delay is 36 ms and bandwidth is 3.2 Mb/s). In `Network Fail`, the signal is seriously weak (package or message delivery delay lengthens to 451 ms, bandwidth drops to 12.0 Kb/s). In `Flight Mode`, WiFi is closed at software level by the OS, and GPS works normally.

The **reason** for choosing `Network Fail` and `Flight Mode` as representatives for special contexts is that our empirical study shows they are the two major types of special contexts. The former accounts for 31.8% (7 out of 22), the latter accounts for 13.6% (3 out of 22) of issues manifested under special context.

We also designed a special type of runtime context, `Non-background`. We designed this context because in our experiment we observed that it can provoke more no-sleep issues. In `Non-background`, the network and GPS work ordinarily, however, we do not input a press of *Home* button to the device after `EXECUTION` stage (the stage-division for test cases will soon be explained in Section 5.3). That is, the test case does not have `BACKGROUND` stage, and straight goes to `SCREEN-OFF`.

5.2 Steer the Test Direction On-the-fly

Our framework steers test direction dynamically based on test history. Algorithm 1 shows details of our steering scheme. The rationale behind it is this: when an energy issue is detected, it implies that this type of input sequence and runtime context may have a greater opportunity than usual to provoke energy issues since it did trigger an energy issue. Hence, our framework will generate slightly more of this type of test cases for larger chance of confronting energy issues.

Data for the algorithm. The candidate input sequences and runtime contexts are designed based on the approach we demonstrated in Section 5.1.1 and 5.1.2. We present them in the data structures of $WeightedInputSequences$, *Random InputSequences* and *Runtime Contexts*. N is the number of candidate runtime contexts. In our test, we devised 4 runtime contexts (including `Non-background`), so $N = 4$. p_{wgt} and P_{ctx} are the steering parameters for concretely guiding the test. p_{wgt} is the probability of choosing a **weighted** input sequence for the upcoming test case. In practice, we initialize it as 50%, so the first test case has a chance of 50% to run with a weighted

[11] https://developer.android.com/studio/test/monkey.html.

Algorithm 1: Steer the test direction on-the-fly

Data:
$WeightedInputSequences = \{(sequence_i, weight_i)\}$;
$RandomInputSequences = \{sequence_j\}$;
$RuntimeContexts[N] = \{context_k\}$;
$0 < p_{wgt} < 1$, $P_{ctx}[N] = \{0 < p_k < 1\}$;
$\Delta_{wgt}, \Delta_{cxt}$;

1 Preparation before testing;
2 Start a test case, clear unrelated threads and data, launch app from root screen;
3 #----------Select Input Sequence and Runtime Context----------#
4 Determine the type of input sequence, and the type of "weighted" has a probability of p_{wgt} to be chosen;
5 **if** *the determined type is "weighted"* **then**
6 | Select an unexplored sequence with highest $weight$ in $WeightedInputSequences$;
7 **else**
8 | Randomly select one sequence from $RandomInputSequences$;
9 **end**
10 Select one context from $RuntimeContexts$ with its corresponding probability;
11 #----------#
12 Run app with the selected input sequence and runtime context on device, and meanwhile measure power consumption;
13 Identify energy issues from power trace;
14 **if** *there exists an energy issue* **then**
15 | Add the energy issue to database;
16 | #----------Update steering parameters----------#
17 | **if** *the issue is incurred by a "weighted" sequence* **then**
18 | | **if** $p_{wgt} \leq wgt\_up\_threshold - \Delta_{wgt}$ **then** $p_{wgt} := p_{wgt} + \Delta_{wgt}$;
19 | **else**
20 | | **if** $p_{wgt} \geq wgt\_down\_threshold + \Delta_{wgt}$ **then** $p_{wgt} := p_{wgt} - \Delta_{wgt}$;
21 | **end**
22 | **switch** *which context triggers the energy issue* **do**
23 | | **case** *e.g.* $context_k$
24 | | | **if** $p_k \leq cxt\_up\_threshold - \Delta_{cxt}$ ***and*** *there are n elements (except* p_k*) in* P_{ctx} *that are greater than or equal to* $cxt\_down\_threshold + \Delta_{cxt}$ ***and*** $n > 0$ **then**
25 | | | | $p_k := p_k + \Delta_{cxt}$;
26 | | | | Decrease those n elements individually by Δ_{cxt}/n;
27 | | | **end**
28 | | **end**
29 | **endsw**
30 | #----------#
31 **end**
32 **if** *time expires* **then End Testing** and generate test reports;
33 Go back to **line 2**

input sequence, and we will update and refine p_{wgt} dynamically during the entire testing. On the other hand, one element p_k in P_{ctx} represents the probability of choosing $context_k$ in *RuntimeContexts*. We will also update P_{ctx} at runtime. The summation of elements in P_{ctx} is bound to 1.

Δ_{wgt} is the increment used to increase or decrease p_{wgt} to renew p_{wgt}. Δ_{cxt} plays the same role for P_{ctx}. The larger Δ_{wgt} and Δ_{ctx} are, the more aggressively we tune the test direction.

Details of the algorithm. We first prepare data and initialize parameters (p_{wgt}, P_{ctx}, Δ_{wgt} and Δ_{cxt}). We then start a test case, clear unrelated threads and data, and launch the app from root screen. Next, we decide the

type of input sequence; weighted input sequences have a probability of p_{wgt} to be chosen. If the chosen type is "weighted", we then select an unexplored sequence with the highest *weight* in $WeightedInputSequences$. Otherwise, we randomly select a sequence from $RandomInputSequences$. Likewise, for runtime context, we select one from RuntimeContexts with its corresponding probability.

We then feed the app with the selected input sequence and runtime context, and measure the device power. The power trace will be analysed to confirm whether an energy issue occurs. If there exists an energy issue, it implies that this type of input sequence and runtime context may be profitable for provoking more energy issues, our testing framework then steers lightly in this direction. Specifically, if it is triggered with a weighted input sequence, we increase p_{wgt} by Δ_{wgt}. And after being increased, p_{wgt} should not exceed $wgt\_up\ \_threshold$. If it is a random input sequence, we decrease p_{wgt} by Δ_{wgt}. Also, we keep $p_{wgt} \geq wgt\_down\ \_threshold$.

An analogous approach is applied to refining P_{ctx}. We check under which runtime context (e.g. $context_k$) the issue occurs, then increase its testing probability (e.g. p_k). However, the precondition is that there should be at least one element (except p_k itself) in P_{ctx} that are no less than $cxt\_down\ \_threshold + \Delta_{cxt}$, because on one hand, we intend to rebalance the probabilities, and on the other, we should let all contexts have at least a possibility of $cxt\_down\_threshold$ to be tested.

5.3 Identify Energy Issues from Power Trace

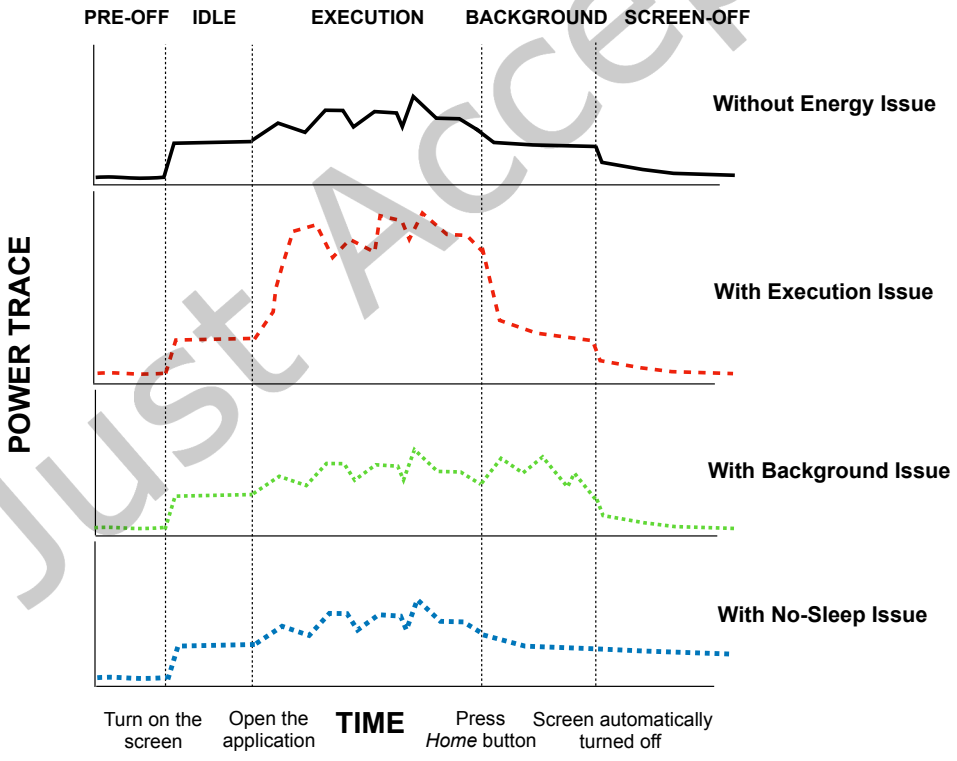


Fig. 5 An illustration of power traces with energy issues.

We divide the power trace into five stages, i.e. PRE-OFF, IDLE, EXECUTION, BACKGROUND and SCREEN-OFF. This division can help us identify three types of energy issues: execution issues (including issues caused by unnecessary workload and excessively frequent operations), background issues (i.e. wasted background processing) and no-sleep issues. Figure 5 shows an illustration of power traces with these three types of energy issues.

And it is worthy to mention the detail that, the sampling rate of our measurement is 100 Hz. Thus the granularity of time of the power trace is one-hundredth of a second.

PRE-OFF stage is the beginning stage where the device is powered but the screen is off. Then, the test case will be transferred to IDLE stage by turning on the screen. To enter EXECUTION stage, the subject application will be opened and run with a certain input sequence and runtime context, which are selected as shown in Section 5.2. After EXECUTION stage, the application will be fed with a press of *Home* button to enter BACKGROUND stage. The final stage is SCREEN-OFF stage, which begins when screen is supposed to be turned off automatically, however, part of energy issues keep the screen on even at SCREEN-OFF stage, eating battery power badly.

5.3.1 Identifying Execution Issues. For execution issues, as we discussed in Section 1, evaluating the necessity of app workload is crucial for identifying them. Specifically, we employ the Dbscan clustering algorithm [27] (Density based spatial clustering of applications with noise) to fulfil this purpose. The objective of Dbscan is to classify multidimensional data points into three groups, namely, core points, border points and outlier points. After clustering, the data points should have the following properties: 1) For a core point, the number of its neighbours (the points within a range of ε from it) is no less than a certain value, $MinPts$. Generally speaking, the core points are "quite close and gathered". 2) For a border point, its neighbours are less than $MinPts$, but it is a neighbour of at least one core point or another border point. 3) For an outlier point, its neighbours are less than $MinPts$, and it does not have either a core or a border neighbour.

We treat each test case as a data point, and treat test cases in the same app category as a data set for clustering. The dimensions of each data point we employed for clustering are l_{chpp}, n_{chpp}, μ_{chpp}, μ_{exe}, which are all extracted from power trace of EXECUTION stage of each test case. l_{chpp} is the total length of continuous-high-power periods. Continuous-high-power period is when power continuously exceeds a certain threshold longer than a certain length. In practice, we define the "high-power" threshold as the value of $mean(all) + std(all)$. all is the set of all power samples collected for this app category so far. And the time length is set at one second.

n_{chpp} is the number of these periods. μ_{chpp} is average power of these periods. μ_{exe} is average power of the entire EXECUTION stage.

Dbscan then classifies the test cases into those three groups. We label test cases in core and border groups as "normal", and the ones in outlier group as suspects for suffering from execution issues.

The motivation of this approach to probing for execution issues is this: usually "normal" energy use is sufficient to guarantee quality of user experience (QoE) of apps, outlier-level energy use is suspiciously problematic and unnecessary. And different app categories usually have distinct "normal" energy use (e.g. games *vs.* productivity apps). So we handle different app categories separately as shown above.

As shown above, we use app category as a heuristic to determine the necessity of workload. Our evaluation (in Section 6) demonstrates the high efficacy of this heuristic. Finer-grained and better categorization of apps may further enhance the efficacy. Here is an application scenario of our framework: the testers can treat the app under test and its competing apps on the market as a comparison group; Outlier-level energy use in such a setting likely indicates energy issues that are worth further investigation and fixing.

Note that, the ultimate clustering model is available only when all test traces are collected. So at the beginning of testing, such a model is inaccessible since we have no power traces to build it. That is, we have no clustering model for identifying execution issues at the beginning of testing. So, we need to decide, from which point during testing, we should start building a model for temporary use.

To make our design reasonable, we did a pilot study by running 100 randomly-chosen test cases. We extracted the key features from the power traces and visualized the features. We found two "obvious" outliers using Dbscan. With further manual verification, both outliers were confirmed to be real issues.

As a density-based clustering algorithm, Dbscan may generate biased-models when outliers are of significant proportion in the dataset. In our problem domain, the number of outliers (abnormally high energy use) is naturally low, as shown in our pilot study and in our final evaluation (Section 6). Hence, in our experiment, we at first collected 50 power traces to bootstrap the model building process and then iteratively added new power traces to the dataset. As the dataset grows, the model becomes more accurate and powerful for identifying outliers. At last, we use the final model to recheck all power traces for issue detection.

Algorithm 2: Find the appropriate ϵ (based on [26])

```
# Input
data set of size n
window = wd
# Output
ε for the Dbscan algorithm
### Find distances to the nearest k neighbours for each points
for i = 1 to n do
    for j = 1 to n do
        d(i, j) = find distance (point_i, point_j);
        find distances to the nearest k neighbours;
    end
end
sort the k * n distances in an ascending order and plot them out as a curve;
### Find the critical change in the curve, and ε corresponds to it
for i = 2 * wd to k * n do
    compare the previous two windows of distances by rank-sum test;
    if the two windows are unequal then
        the mean value of the distances in the two windows is the proper ε;
        break;
    end
end
```

Later, our evaluation on 89 apps (involving 35600 test cases) shows that only 1.1% test cases are outliers. Our framework detected 47 candidate execution issues from those 1.1% test cases. Only three (out of the 47) are false positives, indicating the high reliability of this approach. In contrast, current technology [17] detected 3 candidate execution issues from 30 apps, and still one of them is a false positive.

5.3.2 The determination of ϵ and MinPts. For easy explanation, the above discussion did not mention the determination of ϵ and *MinPts*. Here we explain it in more details.

MinPts is not difficult to evaluate. The previous section shows that in the domain of energy issue detection the number of outliers is naturally low (usually below 10% of all data points). We thus set *MinPts* at 10% of the number of all data points, to avoid classifying small groups of outliers as normal points. During our testing, ϵ

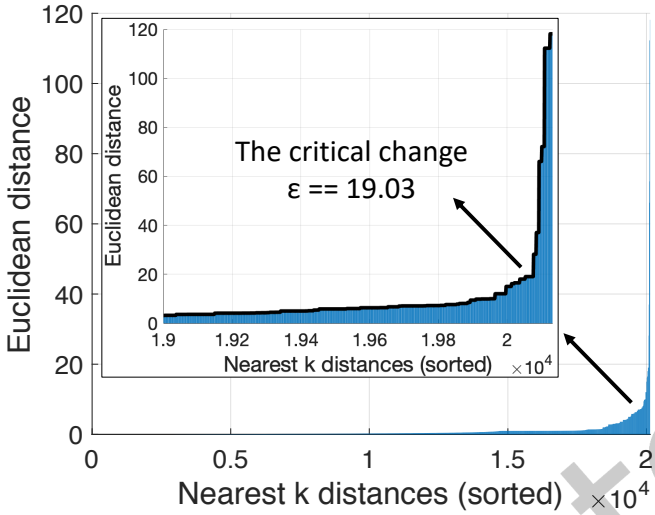


Fig. 6 The example of finding ϵ for the app category of communication.

is dynamically adapted to the current data set. ϵ varies according to the different app categories, and changes iteratively with the enlarging dataset.

So given an already-collected dataset of a certain app category, we employ Algorithm 2 to automatically find the suitable ϵ.

The basic idea of the algorithm is to find the value of ϵ that can generally indicate the density of the data points. Concretely, the algorithm first calculates the distances to the nearest k neighbours for each point, then sorts all of these distances in an ascending order, and further plots them out as a curve. Finally, the critical change in the curve is the proper ϵ.

We set k value at 10 in practice since larger k does not change the proper ϵ dramatically.

Figure 6 shows an example of the critical change in the curve. We go through curve from left to right, and compare two sequential windows of distances in the curve. If the two windows of distances are different, then here (the two windows) is the location of the critical change, and we use the mean value of the distances in the two windows as the proper ϵ. Otherwise, we continue searching for it. And we employ rank-sum test [81] to conduct a statistical comparison between the two windows. Rank-sum test is dedicated to evaluating the equality of two sets of values. In practice, we define the size of the window as 3.

Applying the above approach, we also find the ϵ for other app categories, as presented in Figure 7. And Figure 8 demonstrates the clustering result for communication apps based on the corresponding ϵ.

5.3.3 Identifying background and no-sleep issues. If the app is free from background issues, the power trace in `BACKGROUND` stage is supposed to be similar to that in `IDLE` stage. We thus compute the dissimilarity value of the two traces. If the value is above a certain threshold (40% in our experiment), we label this test case as a candidate for a background issue.

We identify the no-sleep issues in the same way. We compare `PRE-OFF` with `SCREEN-OFF`. If the dissimilarity outstrips a certain threshold (50% in our experiment), we speculate this test case is suffering from a candidate no-sleep issue.

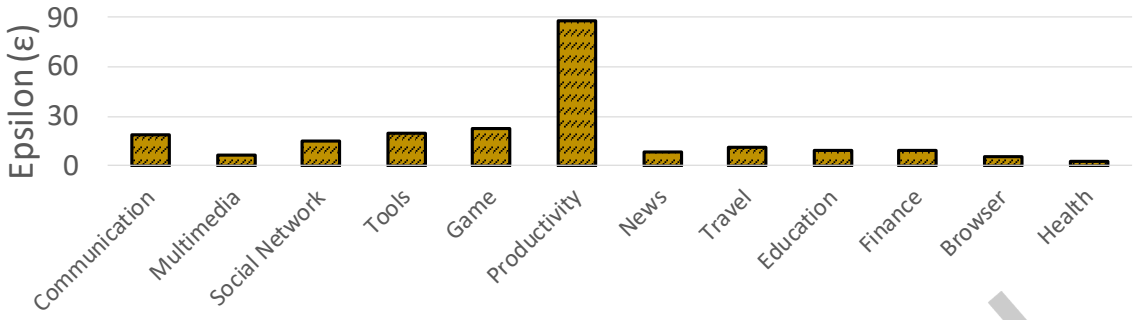


Fig. 7 The ϵ for different app categories.

5.3.4 The length of test cases. Our testing framework does not require the same length of the power traces, because our employed features (e.g., l_{chpp}) can help energy issues manifest themselves no matter in a long test case or a short test case. For example, in a short test case, the energy issue may cause less n_{chpp} but higher μ_{exe}. On the other hand, in a long test case, the energy issue may cause lower μ_{exe} but more n_{chpp}. We set a time threshold for each test case at 3 mins. But the test case could be terminated when it reaches the end of EFG or the app crashes.

5.4 Manual verification

After the candidate issues are found, we manually verify whether they are actual energy issues. We adopt three criteria for distinguishing a real issue from a false positive. First, the energy cost is not from the OS. Second, the energy cost is larger than normal cases by at least 10%. These two criteria are basic, we further apply the third criterion to better filter out the false positives.

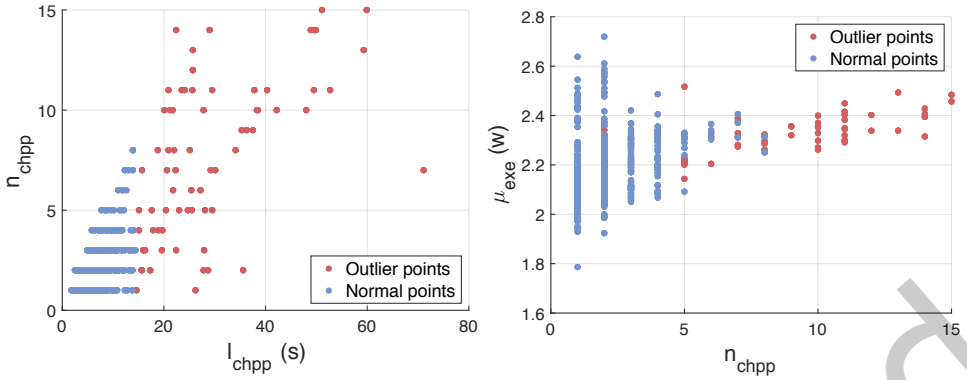
The third criterion is that, user experience can be improved after removing the workloads. Figure 9 shows how we apply such criterion in practice. We first conjecture the outcome of the complete removal of the workloads. If the removal will benefit UX, then the workloads are an energy issue. Otherwise, we further check whether it is possible to minimize the workloads without harming the functionality. If it is possible, then the candidate is determined as an energy issue, because we can use less energy to realize the same functionality. If it is impossible, then there is no issue. If it is not sure, we then conservatively consider it as a false positive as well.

We take two examples to illustrate the above procedures.

For example, an app is downloading large files for functional purposes, such as maps in games, which causes abnormally high energy cost. In this case, there is no easy solution to save battery power since users may accept the expensive downloading process (removing it will affect app functionality). We then identify it as a false positive (*Output#3*).

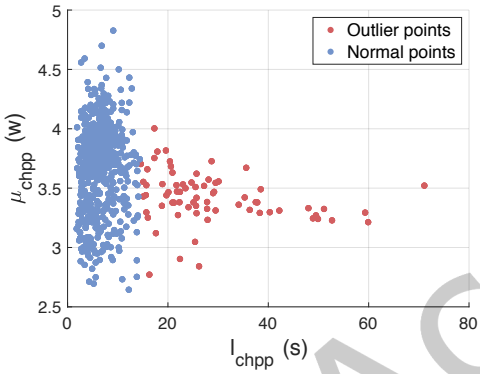
For another example, for the issue in Leisure (as shown later in Figure 17), GIF animations are played when they are invisible. The animations in such cases do not generate user-observable benefits. But completely removing them will threaten UX. So we analyze the source code and attempt to freeze the animations when they are out of screen, to minimize the workloads. We then measure the energy consumption of the test case w/ and w/o our code refactoring under the same runtime context. The result shows that our solution saves 25.9% of the energy consumption without changing any functionality. So such issue is identified as a real energy issue (*Output#2*).

Besides, it is also worthy to notice that, our testing framework already has the capacity of manifesting and detecting the issues caused by the system and configuration changes. It is because our well-designed,

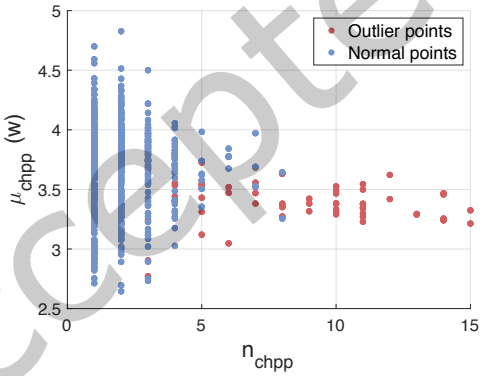


(a) n_{chpp} and l_{chpp}

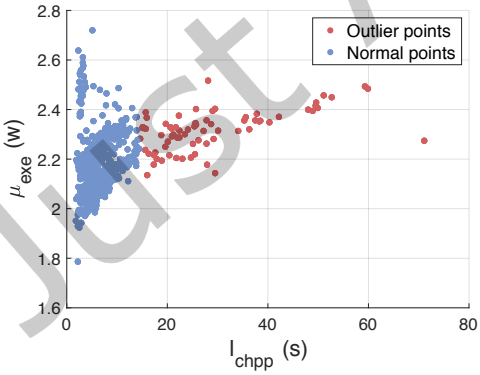
(b) n_{chpp} and μ_{exe}



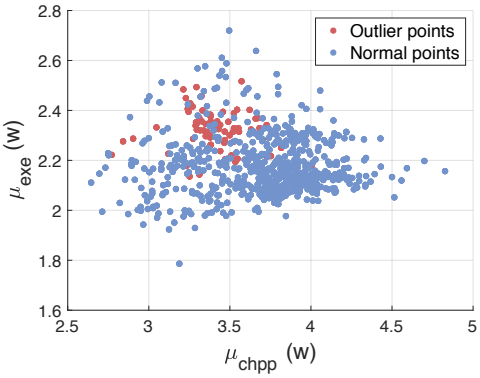
(c) μ_{chpp} and l_{chpp}



(d) μ_{chpp} and n_{chpp}



(e) μ_{exe} and l_{chpp}



(f) μ_{exe} and μ_{chpp}

Fig. 8 The clustering result for *Communication* apps, which is presented according to different pairs of the four features for a full picture. Each point is a test case.

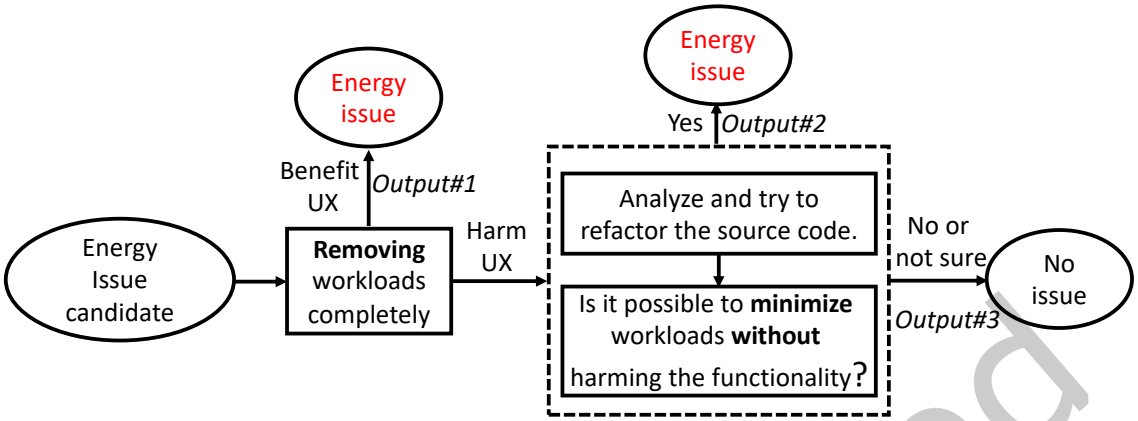


Fig. 9 The flowchart of manual verification.

diversified and effective input sequences and runtime contexts are sufficient to pinpoint issues caused by OS and misconfigurations.

Recall that, when we manually verify the candidate energy issues, we first check whether the issue is caused by OS. If it is, we label it as "no issue". That is, we filter out the system issues. It is because the market object of our testing framework is app developers, who form a huge potential market (Google Play: 600,000 apps; App Store: 650,000 apps [57]).

But this does not mean our framework cannot detect issues caused by the system and the configuration changes. Actually, among the eight false positives detected by our test (see Section 6.2), there are five caused by the system and configuration changes.

Note that, the five "false positives" mean that they are energy issues but not APP energy issues. It is also worthy to mention that, we detect 83 app energy issues. Hence, there are naturally more app energy issues, justified by [57] and our work.

Our testing framework is able to serve OS developers in the same way as how it serves app developers, as shown in Section 6.4. Our testing framework even does not need to be extended to fulfil this purpose. And we will enable system-issue-detection for OS developers to help combat system energy issues.

6 EXPERIMENTAL EVALUATION

In this section, we first introduce our experimental setup. Then we evaluate our testing framework on various aspects, such as its efficacy in detecting energy issues, its comparison with the state-of-the-art, etc. The result shows that our testing framework largely outperforms current technology, showing the benefits both of our sound empirical study and our dynamic targeting techniques.

6.1 Experimental setup

We employ the Odroid-XU4 development board[12], whose processor has four big cores with a frequency of 2 GHz and four small cores with a frequency of 1.3 GHz. The board possesses a powerful 3D accelerator, Mali-T628 MP6 GPU. The high capacity of Odroid-XU4 board guarantees its performance for most applications on the market. It is also equipped with a power monitor, Smartpower2[13], to measure the real-time power consumption. The

[12]https://wiki.odroid.com/odroid-xu4/odroid-xu4
[13]https://www.odroid.co.uk/odroid-smart-power-2

Table 6 Examples of detected energy issues.

Application	Category	Issue Type	Activity or Symptom	Hit Rate[1]	Context or Non-background	Energy Waste	Reported before?
Leisure	News	Execution	3 GIFs playing on one page	1.0%	Normal	25.9%	No
GPS Status	Travel	Execution	Not obvious	47.0%	Normal	15.3%	No
BatteryDog	Tools	Execution	Editing lengthy text	1.0%	Normal	30.8%	No
Rocket Chat	Comm.	Execution	Connect to server	1.0%	Normal	12.8%	No
Chess Clock	Tools	Background	Device heated up	100.0%	WiFi Fail	59.2%	No
Vanilla	Multimedia	No Sleep	Enqueue many tracks	59.0%	WiFi Fail	242.8%	No
cgeo	Travel	No Sleep	App get stuck	2.0%	Flight Mode	179.4%	Yes
AntennaPod	Multimedia	No Sleep	Keep popping up messages	1.0%	Non-background	184.4%	Yes

1. Hit rate here is the percentage of test cases detected having the energy issue in that app.

sampling rate is 100 Hz. To assess its measurement variability, we randomly choose 20 test cases and run each for 10 times. We thus collect 10 records of average power consumption for each test case. We employ coefficient of variation (C_v) to indicate the variability. Specifically, $C_v = \frac{\sigma}{\mu}$, where σ and μ are the standard deviation and mean of the 10 records for each test case. At last, the mean of the 20 C_vs is about 0.8%, meaning a low measurement variability. Due to these rich and solid features, the Odroid board is widely employed in the field of energy optimization for mobile devices [67, 84, 90].

We use Android 4.4 KitKat as our target OS. Android is open-sourced and it captures around 71.83%[14] of the worldwide mobile OS market by Mar 2021. We evaluate our framework on 89 app subjects, which are arbitrarily selected from the 98 app subjects in our empirical study. 27 ones (of the 89) have energy issue reports, 62 do no, as we introduced in Section 2. The **considerations** of employing these 89 apps for the evaluation are these: 1) Our framework is inspired by issue reports from the 27 apps, so it may be inapplicable to new apps, we thus employ the 62 subjects without issue reports as **new** apps to evaluate the generality of our framework. 2) Referring to the issue reports, we also can check whether our framework is capable of detecting unreported issues. 3) These apps are popular, well-maintained and of much higher quality than the apps adopted by previous research.

Our total testing time for the 89 app subjects is 2373.3 hours, i.e. 98.9 days, which were evenly spent on each app. (i.e. 1.11 days for one app). Note that, the time of preparing weighted input sequences is also included, which takes 18.8% of the entire testing time.

6.2 The efficacy of our testing framework

6.2.1 The accuracy and effectiveness. The experimental result shows that our test detected 91 candidate energy issues, among which we manually confirmed **83 real energy issues**. 22.9% (19 out of 83) of these issues are from the 27 old apps, 77.1% (64 out of 83) are from the 62 new apps. It is worthy to mention that 14 out of the 19 issues were unknown previously. After manual verification, we found **8 false positives**. Table 6 shows 8 examples of the detected energy issues. For instance, in `Leisure`, three animated GIFs are loaded and are played at the bottom of a certain page even though they are invisible to users most of the time. This execution issue wastes 25.9% energy use. It can be fixed by freezing the animation when the GIF pictures are not shown on the screen. For another example, when `Chess Clock` is not in use and backgrounded, the device will heat up from 41.2°C to 60.9°C due to the inefficient and long utilization of CPU. The average power of this issue is 59.2% higher than that of the `IDLE` stage.

Figure 10 shows the energy waste of detected energy issues. Energy waste is calculated using Equation (4).

[14]http://gs.statcounter.com/os-market-share/mobile/worldwide

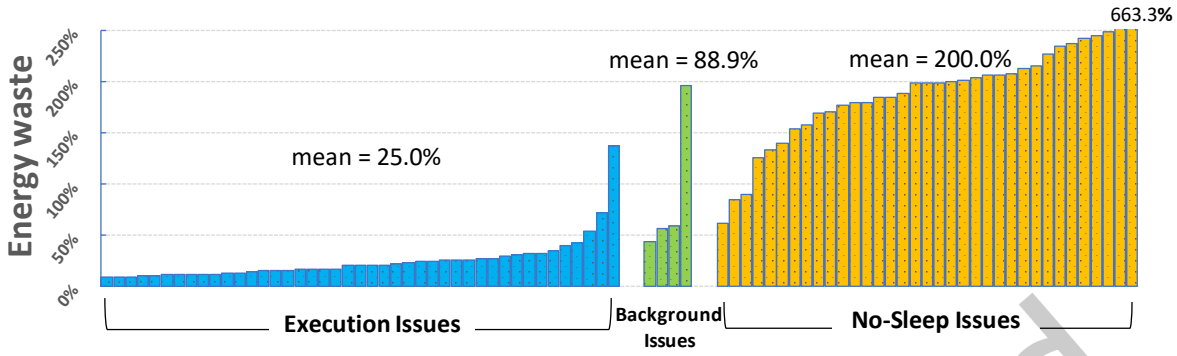


Fig. 10 The energy waste of detected issues of different types

$$w = (\frac{e_x}{e_n} - 1) \times 100\% \tag{4}$$

w is the energy waste of the issue, e_x is average power of the corresponding stage in the test case with the corresponding issue (`EXECUTION` stage for execution issues, `BACKGROUND` stage for background issues, `SCREEN-OFF` stage for no-sleep issues). e_n is "normal" energy cost. We define "normal" energy cost individually for different issues. For background and no-sleep issues, we adopt average power at `IDLE` and `PRE-OFF` stage as normal cost, respectively. For execution issues, we use mean value of average powers of `EXECUTION` stage in test cases in the same app category, as normal cost.

The experimental result shows that the energy waste of execution issues is 25.0% on average and up to 137.6%; the energy waste of background issues is 88.9% on average and at maximum 196.2%; the values for no-sleep issues are 200.0% and 663.3%, respectively. Overall, the average energy waste of all the issues is 101.7%.

90.4% (75 out of 83) detected energy issues in our test are **newly-reported**. Without our tests, these serious energy issues might have never been detected even though they cause serious battery drain.

Specifically, the detected issue that shares the following criteria with any one of the 189 issue reports, is deemed as a previously-known issue.

- They have the same input sequence.
- They have the same runtime context.
- They have the same symptoms diagnosed by the same tool.
- They have similar energy wastes.

For example, we treat AntennaPod issue 1110 and one of our detected issues as the same issue because of the below.

First, they have the same input sequence: 1, open the app, 2, listen to a video podcast, 3, switch AntennaPod to background, 4, use other apps or leave the phone to screen-off automatically.

Second, they are both under flight mode.

Third, the same tool of Wake Lock Detector [11] shows that the app is active all the time.

Fourth, the battery drops quickly. The user reported that the battery dropped from 80% to 10% in a few hours. Our test shows that the average power of the app is 2.24 watts, twice as high as the normal power consumption, which can drain battery in a few hours.

On the other hand, 95.8% (181 out of 189) energy issues in our empirical study are not listed in the issues detected by our test. The major reasons are as follows: Firstly, our standard of determining energy issues is much

higher than that of developers. The issues detected in our test have outlier-level impacts on energy use; the energy wastage is usually above 10.0%. However, many issues detected by developers only cause small transient workloads, energy overuse can hardly reach 10.0%. For instance, Firefox issue 1057247 lets app re-fetch the failed favicon every 20 min, which is believed costly. So developers reduce the re-fetching frequency to conserve energy. For another example, VoiceAudioBookPlayer issue 299 makes the app scan material files everytime the user starts and leaves app. Developers then designed a smarter scanner to lessen the number of scans. However, these issues can not be noticed by our framework because they have very marginal influence on the metrics (i.e. l_{chpp}, n_{chpp}, μ_{chpp} and μ_{exe}) we chose to identify execution issues. Secondly, a number (34.3%, 62 out of 181) of the issues are not reproducible, our test cannot trigger them either. Thirdly, the variety of input sequences and runtime contexts in our test is not large enough to cover all of them due to the time limit.

The first and third reasons also shed light on **false negatives** of our test. The first reason implies that the issues that draw developers' attention but consume non-significant power (compared with our detected issues) can be missed by our test. It is because only outlier-level energy cost is deemed as an energy issue in our test. Loosening this strict criterion can help significantly reduce false negatives but may lead to false positives. The dilemma is caused by the difficulty of objectively and automatically judging the necessity of workloads. There is no general solution for this problem yet. Nevertheless, our framework can be flexibly configured to detect more energy issues if users are willing to tolerate some false positives. W.r.t. the third reason, since we were testing 89 apps, one day's test for one app results in three months' test for all apps. Developers can test their own app more comprehensively to realize larger coverage.

6.2.2 The influence of the key parameter of ϵ. We further investigate how the key parameter of ϵ influences the false positives and false negatives of our testing framework.

We scale the ϵ from 0.1 to 1 of the optimal ϵ. And the optimal ϵ is identified using Algorithm 2.

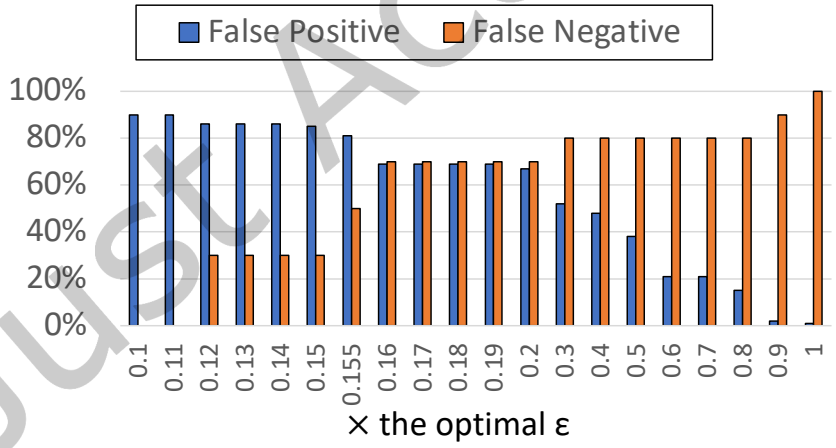


Fig. 11 The influence of ϵ on false positives and false negatives.

The results are shown in Figure 11.

Specifically, to evaluate ϵ's influence on false positives, for each ϵ value we randomly sample 100 issue-suffering test cases detected according to the ϵ value and manually verify the false positives. We then calculate the percentage of the false positives in the 100 test cases.

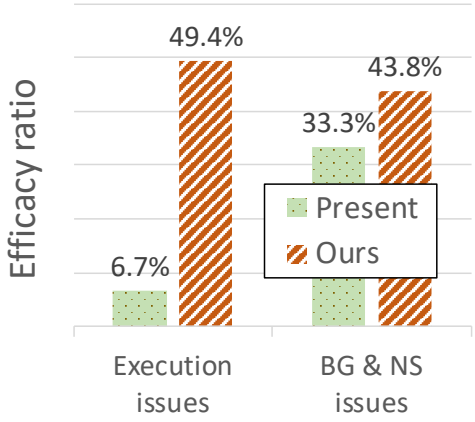


Fig. 12 Comparison on issue-detection efficacy (r_s^d) with the present technology.

To evaluate ϵ's impact on false negatives, we first prepare ten verified false negatives as ground-truth, and then for the each ϵ value we calculate the percentage of the undetected false negatives out of the ten as the result to assess the influence.

So the results indicate that, increasing ϵ will increase false negatives and decrease false positives. We cannot keep them both low. The fundamental reason is that, it is very difficult to judge the necessity of the "small" workload. But our approach does well in judging outlier-level energy issues.

In our practice, we choose to tolerate a high rate of false negatives and benefit a low rate of false positives.

6.2.3 Comparison with the literature, and the evaluation on a new device. We now compare the efficacy of our testing framework with current technology [17]. We use the efficacy ratio (r_s^d), i.e. the ratio of the number of detected issues to the number of subject apps, to indicate the efficacy of a framework. As shown in Figure 12, for execution issues, their r_s^d is 6.7% (2 issues out of 30 apps), ours is r_s^d is 49.4% (44 out of 89); for background (BG) and no-sleep (NS) issues together, their r_s^d is 33.3% (10 out of 30), while ours is 43.8% (39 out of 89). We combine background and no-sleep issues since the work [17] did not distinguish them. The result confirms that our framework can detect a much larger number of more serious energy issues in higher-quality apps in comparison with the state-of-the-art. This is due to the fact that our work is based on the insightful empirical findings, rather than ungrounded assumptions.

We use Monsoon [6] power monitor to trace the realtime power consumption. As shown in Figure 13, we reconfigure the battery of the phone: we bypass the battery and use the power monitor to power up the phone and gauge the voltage, current and thus power of the phone.

We have two sets of app subjects. The first set contains the 14 old communication apps in the previous experiment. The second set has 10 new communication apps. We run our testing framework on the two app sets separately. We also run the testing under two schemes individually. The first scheme is testing with the steering algorithm (i.e., Algorithm 1), the other is without it. For both schemes, we set the time budgets at 50 hours.

So this experiment, on one hand, evaluates our framework's efficacy on a new device and new&old apps. On the other hand, it also assesses the effectiveness of our steering algorithm.

As displayed in Figure 14, for old apps on Samsung S20 our framework achieves a r_s^d of 78.6% with the steering algorithm, nearly three times higher than that without steering. For new apps, r_s^d of testing with steering is six times higher than testing without steering.

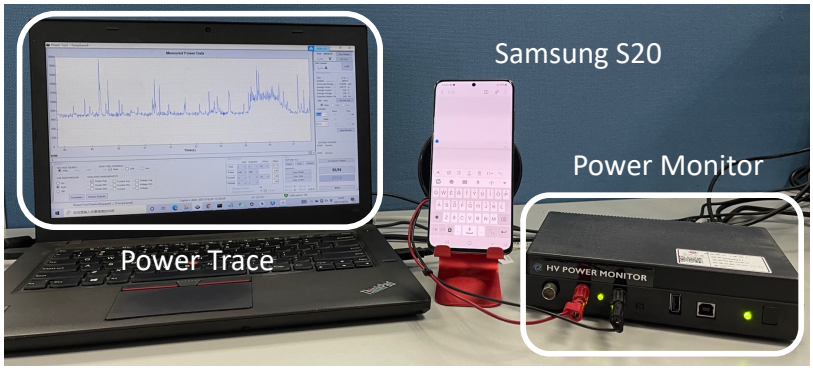
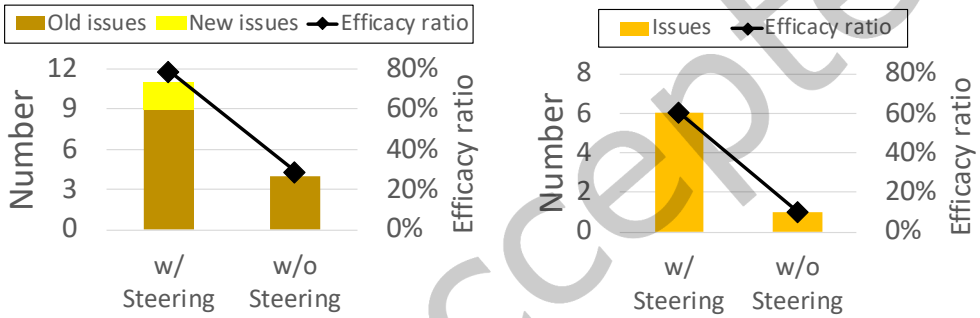


Fig. 13 The implementation of power measurement for Samsung S20.



(a) Evaluation on the 14 old communication apps

(b) Evaluation on 10 new communication apps

Fig. 14 Evaluation on a new smartphone, Samsung S20.

The results also present that the overall r_s^d for the new device (i.e., Samsung S20) is 70.8%. The r_s^d on the old device (i.e., Odroid development board) is 93.2% (49.4%+43.8%). Considering the testing time on the new device is much less than that on the old device, our testing framework still demonstrates satisfying r_s^d.

6.3 Issue Cause and Manifestation

Figure 15 demonstrates breakdown of energy issues of different causes in empirical study and experiment. "Exp." is experiment, "Emp." is empirical study. Our experiment is conducted on both new and old apps, we thus plot them with different colours and patterns. "UW" is unnecessary workload, "EFO" is excessively frequent operations, "BG" is wasted background processing, "NS" is no-sleep, "SW" is spike workload, "RE" is runtime exception. As shown in Section 6.2, the issues detected in experiment are much more costly than those in empirical study. So this result indicates that, no matter for "big" or "small" issues, UW and EFO are always the very significant root causes, which justifies our Finding 1. And these issues are exactly the issues that current technology can hardly address.

Figure 16 demonstrates the number of energy issues triggered under different contexts in our experiment, which captures more detailed manifestation characteristics of energy issues. "Norm" is `Normal`, "Fail" is `Network Fail`, "Flight" is `Flight Mode`, "Non-bg" is `Non-background`. We can see that most execution issues (unnecessary

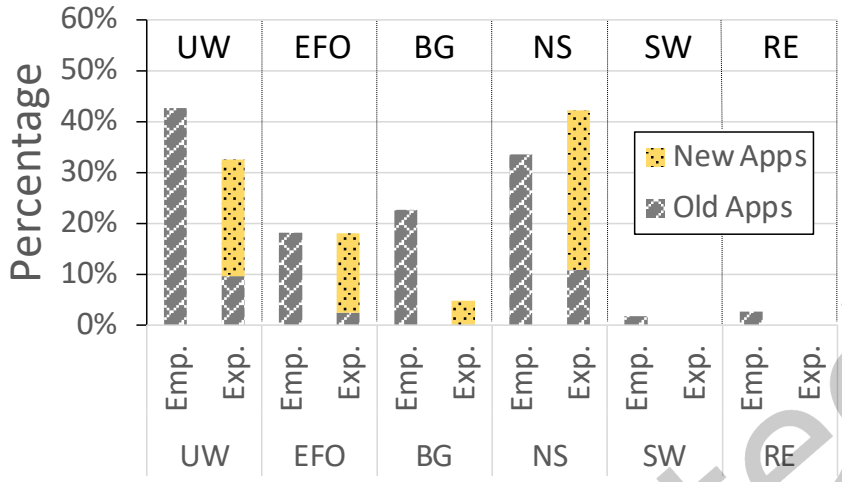


Fig. 15 Issue causes in *empirical study* and *experiment*.

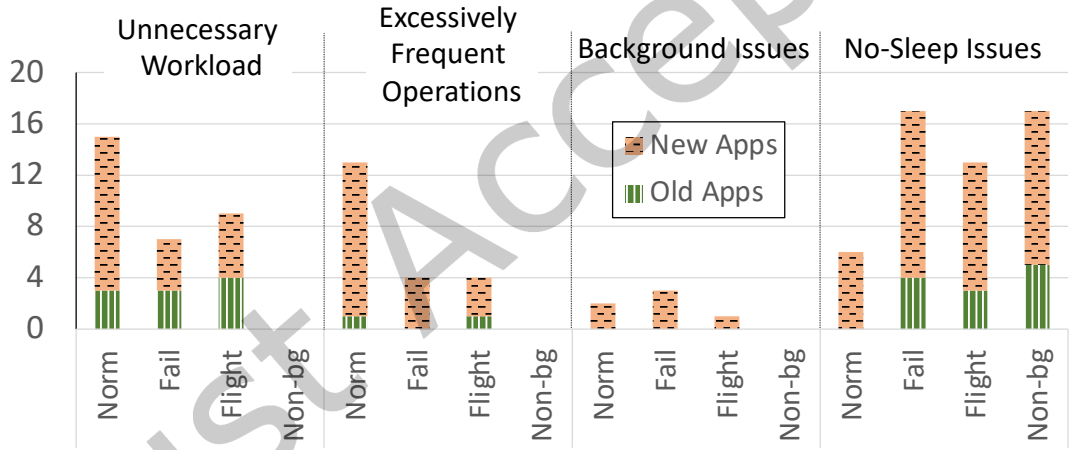


Fig. 16 Energy issues manifested under different contexts.

workload and excessively frequent operations) are manifested in the `Normal` context because many scenarios where issues occur require normal network and GPS context. For example, as we discussed above, the issue in `Leisure` showed up only when those three GIF pictures were downloaded and showing on the page. We only have six (6/83 = 7.2%) background issues, which indicates that OS is competent in clearing potential bad influence of backgrounded apps at `BACKGROUND` stage. However, backgrounded apps may still suffer from no-sleep issues: `Normal`, `Network Fail` and `Flight Mode` provoke 6, 17, 13 no-sleep issues respectively. Furthermore, even though `Non-background` and `Normal` run in the same context, the former tends to provoke more no-sleep issues. In our test, it induces 17 issues. This result implies that special contexts (and `Non-background`) tend to incur anomalous behaviours of apps, such as bad use of wake-lock, and thus cause more no-sleep issues.

Figure 16 also shows that 37.3% (31 out of 83) energy issues can only be triggered under `Network Fail` and `Flight Mode`. This confirms Finding 2: special contexts, such as network fail, hide a significant number of serious energy issues.

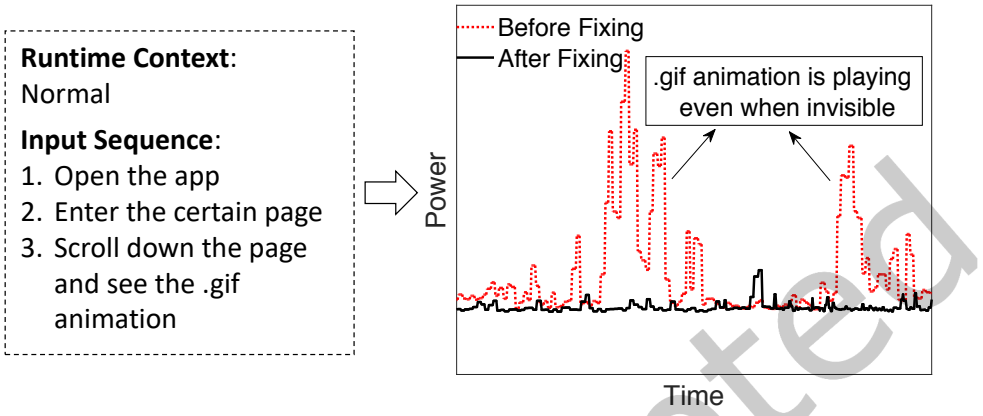


Fig. 17 Before and after fixing the energy issue in Leisure.

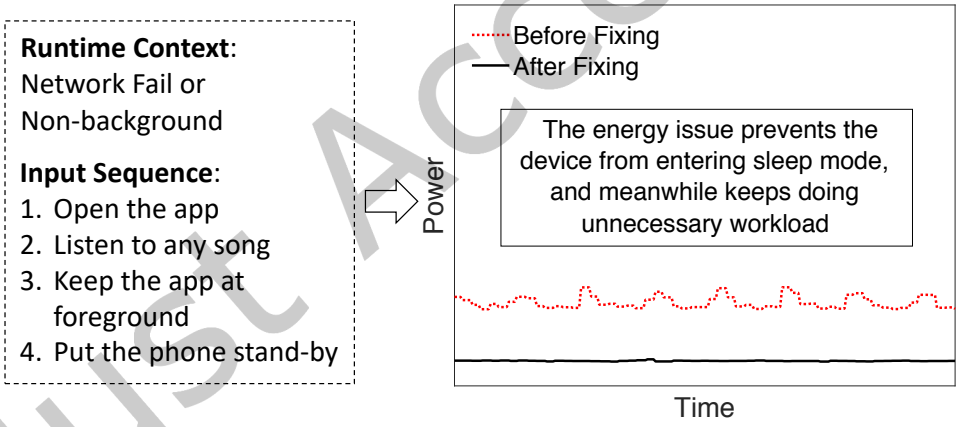


Fig. 18 Before and after fixing the energy issue in Vanilla.

6.4 How our framework benefits developers?

After issue detection, our framework generates a test report to help fix the issue. The report includes:

- The information on input sequence and runtime context, and screen-casting video recording the test case. Using these, developers can analyse issue manifestation and symptoms.
- The visualized power trace of test case. With this, developers will have an intuitive view on power consumption of the issue.

- The method-level execution trace provided by CPU Profiler. The issue reports in the empirical study showed that this information can mostly (83.3%) help find the faulty code.
- The rationale behind testing framework. Developers will understand how the testing framework works.

We conduct a case study, showing how to utilize test reports to diagnose and fix energy issues. We take the execution issue in Leisure and the no-sleep issue in Vanilla (as we listed in Table 6) as two examples. We reproduce the issues, observe the symptoms, check the power traces (the red dash lines in Figure 17 and 18). We then analyse the frequently-called methods to find faulty code, and refactor the code to remove the unnecessary workload and release the wake-lock after use. As we can see, power traces (black full lines in Figure 17 and 18) after fixing are much flatter or lower than those before fixing.

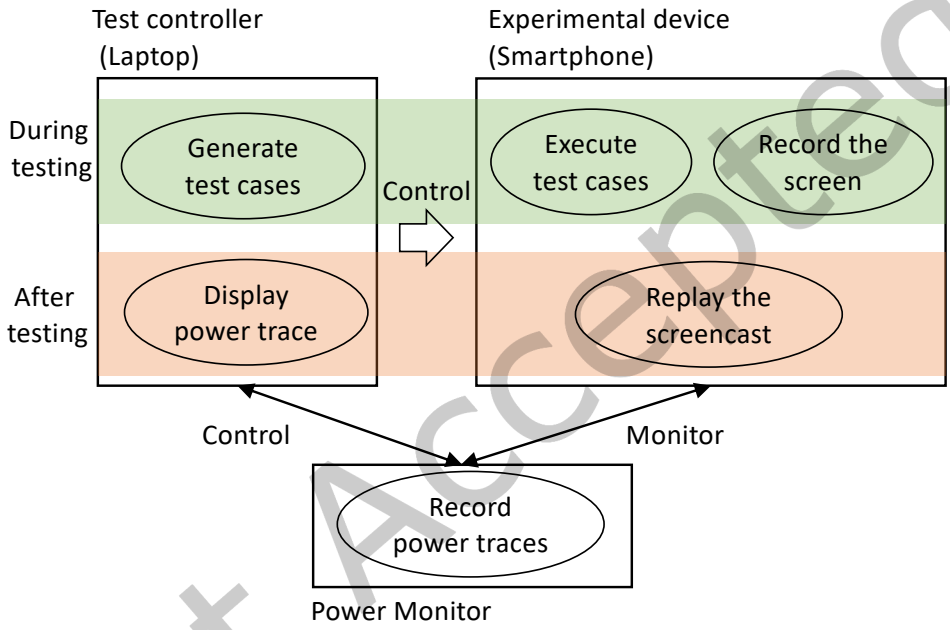


Fig. 19 The hardware and functional structure of our testing framework.

To better illustrate how our testing framework functions, we use Figure 19 to present the hardware and functional structure of our testing framework.

So, during testing, the test controller generates and feeds the test cases to the experimental device based on Algorithm 1. Then, the experimental device executes the test cases. Meanwhile, the screen is recorded.

After testing, when the developer wants to analyse a test case that has energy issues, the test controller displays the power trace of the test case, and the smartphone also replays this test case on the screen for diagnosis.

We also organize a user study to evaluate the usefulness and usability of our testing framework. The usefulness means whether our technology is useful to developers, and whether the developers do need our technology. The usability demonstrates whether our testing is effective and efficient in uncovering energy issues, and helping developers fix the issues.

UX community [5] suggests that a proper number of participants for evaluating usefulness and usability of a product is five. It is because a larger number of participants does not mean better, considering return (e.g., the number of uncovered drawbacks) on investment (e.g., time and money).

We recruit five developers with varied levels of development experience: one with 0.5 to 1 year experience, two with 1 to 2 years, two with 2 to 5 years. We let them run our testing framework and randomly choose two issue reports to reproduce the issues and diagnose the symptoms and try to fix the issues. Afterwards, we ask them to evaluate their individual user experience on our framework and give UX scores of the usefulness and usability. The score ranges from 1 to 5. 1 is "very bad", 2 is "bad", 3 is "acceptable", 4 is "satisfied", 5 is "very satisfied".

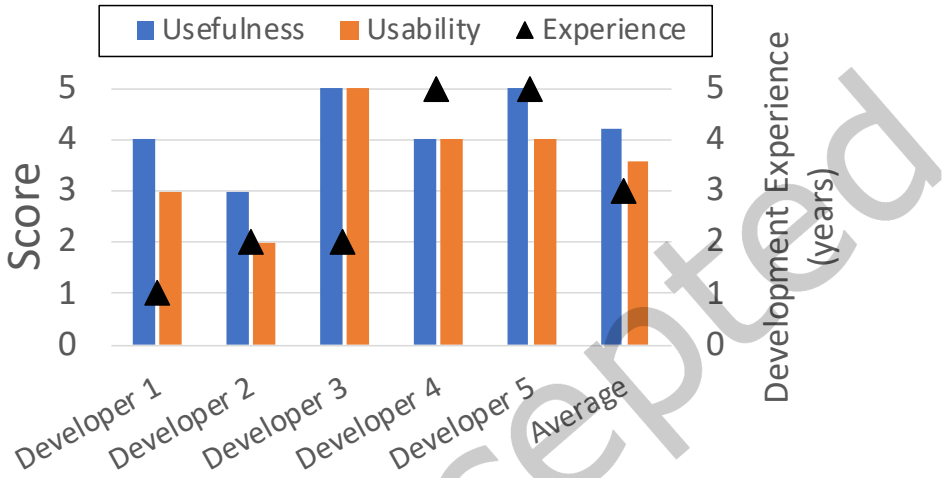


Fig. 20 User study on usefulness and usability of our testing framework.

As presented in Figure 20, our technology provides above-3 usefulness and usability, which is capable of satisfying developers.

We also ask developers about their specific comments on the strengths and weaknesses of our testing framework.

The major strengths are the following.

First, in many cases, the developers are surprised by the high energy-consumption of the detected issues. For example, they did not believe that the GIF animation out of screen is that costly. This tells us that energy use of the apps is the key criteria to judge whether the apps have energy issues. Even though it looks so obvious, our work is the first one to achieve this. Before, researchers either use indirect information (e.g., hardware or resource utilization [54, 56]) to identify energy issues, or use inaccurate information (e.g., E/U ratio [17]) to do it.

Second, reproducing energy issues is very helpful. Again, it looks too evident, but reproducing energy issues is very challenging because of the dynamic and complex runtime context of mobile systems. In contrast, our testing is conducted under strictly-controlled runtime context, the test cases can be reproduced in exactly the same way as it fist time occurred.

The major weaknesses are as below.

First, there is no automatic fix to the detected energy issues. But we discussed in Section 3.3, fixing energy issues is not easy, always requires non-negligible human efforts, more importantly, human intelligence.

Second, the data visualization of CPU profiler is not very easy to read. CPU profiler is a third-party tool, which is provided by Android Studio. We suggest developers using it for localizing the energy issues (see Section 6.4) since this tool can mostly (83.3%) help find the problematic code. But surely, it requires some extent of familiarity with the tool.

7 THREATS TO VALIDITY

There may exist threats to the validity of the empirical study and the experimental evaluation for our testing framework.

For the empirical study, one threat is the representativeness of our app subjects and energy-issue reports. But our empirical study is the largest-scale in the literature w.r.t. both the app number and issue-report number. The most relevant work [54] involves 36 app subjects, but they only investigate 66 issue reports, and we have 36 apps as well, but 200 issue reports. Note that, the selection standards of our app subjects are also much higher than theirs in terms of popularity, maintainability and informativeness.

For the experimental evaluation of our testing framework, the validity may also be subject to the representative of app subjects. But we use the same set of app subjects in our empirical study, which are of higher quality than the most relevant work [17] for the energy issue detection. Note that, the scale of our evaluation is still the largest, containing 89 app subjects. In contrast, [17] only evaluates 30 apps.

The not-large diversity of our experimental devices may also threat the validity of our study. However, our study is conducted both on development board (i.e. Odroid) and the latest commercial smartphone (i.e., Samsung S20). In contrast, [17]'s experiments are purely based on a dated phone (LG Optimus L3, released in 2012).

8 CLARIFICATION

The paper style of "an empirical study + a novel technology". The "an empirical study + a novel technology" researches seem straightforward and heuristic. But they are fundamental, useful and interesting.

We take the work of [56] as an example. In its empirical study, the researchers learn several common issue patterns of performance issues. The two most common patterns are: lengthy operations in main threads ($11/52 = 21.2\%$) and wasted computation for invisible GUI ($6/52 = 11.5\%$).

Hence, their proposed issue-detection technology is "simply" searching for these two patterns in the source code. And the resulting technology achieves a high efficacy in detecting performance issues.

The above issue-detection technology seems simple and heuristic, but it is the empirical study makes it looking simple. The empirical study helps researchers find the "straightforword" direction of the technical research. Without empirical study, researchers even do not have a direction to work on.

Similar to our work, the empirical study makes our technology look straightforward and heuristic. Without empirical study, we even cannot define energy issues (The former research [17] used E/U ratio to define energy issues, which was proved inaccurate and even wrong).

Our empirical study is the largest scale of its kind. We have 200 issue reports. [54] has 66 issue reports. [56] has 70 issue reports.

The number of our app subjects is 36. [54] is 36. [56] is 29. So ours is still the largest.

In addition, we have six RQs. [54] has three RQs. [56] has four. And their RQs are relatively "simple" to analyse, such as manifestations and fixing efforts. In contrast, our RQs not only contain their RQs, but also hard-to-analyse ones, such as the reasons of no fixes.

So, we summarize two core points:

- The number of app subjects in our study is at the same scale with the relevant studies [54, 56]. The number of our issue reports, i.e. the scale of our empirical study, is **two to three** times as large as the relevant studies. More issue reports will reveal more information and insights of the characteristics of energy issues. Also, note that, the quality of our issue reports is much higher than theirs.
- Considering the number of issue reports and the number of RQs, our study requires at least **6.06** times as heavy as the human efforts of the relevant study [54], and **4.29** times as heavy as the other one [56]. Please

note again that, our RQs are also more difficult to analyse.

The importance and relevance of RQ3-RQ6. The following points back up the relevance of RQ3-RQ6 to our paper, as well as the importance to the literature.

- RQ3-RQ6 are well fit to the topic of "Combatting Energy Issues for Mobile Applications" of our paper.
- "An empirical study + a novel technology", "the technology is motivated by **Part** of the empirical study" and "the empirical study and the technology have their own contributions individually": such style of paper organization is highly recognized in the literature, such as [54, 56]. We take [56] as an example. [56] studies four RQs: issue types, issue manifestations, issue-fixing efforts and issue patterns. But their technology for performance-bug-detection is only related to the last RQ (issue patterns).
- RQ3-RQ6 hold their research importance on their own, which is supported by the following facts:
  - In the dimensions of either the app number or the report number, our empirical study is the largest-scale. Also note that, the quality and informativeness of our issue reports are higher than the previous works [54, 56].
  - Regarding the depth of our empirical study, we can compare our study with [56]. [56] studies four RQs: issue types, issue manifestations, issue-fixing efforts and issue patterns. For example, [56]'s answer to RQ1 (issue types) is GUI lagging, energy leak and memory bloat. The proportions of each of them are also given. The authors also explain the types with vivid examples. And the answers to the other RQs are exactly similar. Our research approach **Strictly** follows [56]. Moreover, we emphasize that we have six RQs, which will reveal more helpful and in-depth information.

  Hence, the scale of our empirical study and the informative and high-quality answers to the RQs strongly ensure the contribution and depth of our empirical study.

9 RELATED WORK

Our work is related to many aspects of research work. We first explain the relation between energy issues and other issues. We then discuss about energy issues on three aspects: *understanding energy issues*; *detecting and diagnosing energy issues*; *fixing energy issues and optimizing software*.

9.1 Energy issues vs. other issues

We use Figure 21 to clarify the relation between energy issues and other issues (including performance issues). So actually, energy issues could overlap with many types of issues. For example, a malfunction (a functional issue) in the software could unnecessarily consume a large amount of energy, and cause an energy issue.

So functional issues (including bugs), usability issues (including bugs) and performance issues (including bugs) could all relate to energy issues.

We agree that performance-energy-overlapping issues are an important part of energy issues. And there was previous study [56] involving such issues. But the study only works on characterizing performance issues. And the study labels energy issues as one type of performance issues. It does not go further to analyse the characteristics of such energy issues.

And other researches on energy issues usually focus on very specific examples of energy issues. They do not have a full picture of energy issues. For example, Pathak et al. [66] mainly study the detection of no-sleep energy issues. Liu et al. [55] attempt to develop the technology for detecting wake lock misuses. Xu et al. [83] and Zhang et al. [88] try to detect the overuse of sensors on smartphones.

That is to say, the literature lacks a comprehensive study purely on energy issues, which are a significant type of issues on their own.

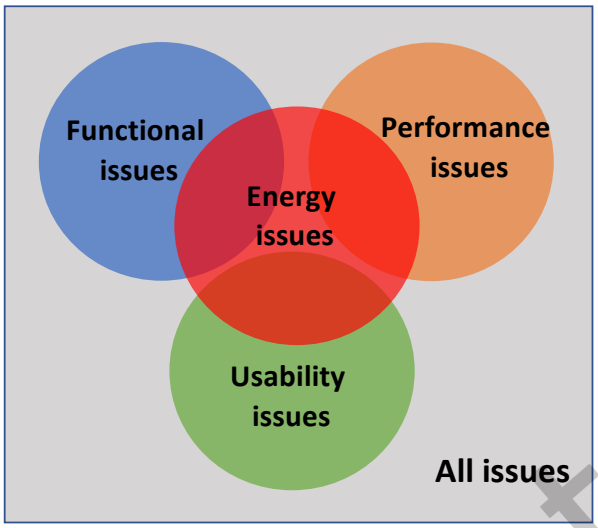


Fig. 21 The relation between energy issues and other issues.

So, our work conducts the most comprehensive empirical study on purely characterizing energy issues. We paint a full picture of the characteristics of energy issues. We analyze energy issues caused by most kinds of hardware components (e.g., screen, CPU, sensors and etc.) and software mechanisms (services, broadcast receivers, alarms, wake-locks and etc.), not constrained to a certain narrow scope of energy issues.

9.2 Understanding Energy Issues

The style of empirical study that mines the data in repositories has been widely applied. For example, to characterize performance issues, a large body of research has been done for PC and server side software [37, 62, 87]. The first empirical study on characteristics of energy issues in the system of mobile device was done by Pathak et al. [64]. They mined over 39,000 posts from four online mobile user forums and mobile OS bug repositories, and studied the categorization and manifestation of energy issues. Xiao et al. [57] conducted a similar survey on three Android forums. The above studies involve issues in multiple layers across the system stack of mobile devices, from hardware, OS to applications. And the issue reports adopted in above studies were mostly proposed by end-users and random developers, who can hardly contribute very insightful understanding of the issues (e.g. root causes). Liu et al. [56] investigated performance issues (including energy issues, as how they treated) reported in issue-tracking systems maintained by the developers who developed the apps. However, the number (i.e., 10) of the studied energy issues is very small compared with our study. The most related study is conducted by Liu et al. [54], however, their work only studies issue cause and fixing efforts. By contrast, we conducted much more comprehensive and insightful empirical study involving issue causes, manifestation, fixing efforts, detection techniques, reasons of no-fixes and debugging techniques.

9.3 Detecting and Diagnosing Energy Issues

Non-energy issues (e.g. security [70], compatibility [45] and performance [63] issues) can be detected plainly by analysing program artefacts. In contrast, to detect energy issues, researchers have to first learn the energy characteristics of mobile devices and apps. Hence, researchers use OS, hardware and even battery features as

Table 7 Technique package. The bold items are our original techniques, the italic are inspired by [17].

Preparation before testing	Steer test direction
• *Candidate input sequences* ◦ *Weighted input sequences* - *System APIs (I/O components)* - **Control-flow operations (CPU)** ◦ **Random input sequences** • **Candidate runtime contexts** ◦ *Normal* ◦ **Non-background** ◦ **Flight Mode** ◦ **Network fail**	• **Balance weighted and random input sequences** • **Balance different runtime contexts** • **Improve efficacy of issue-detection**
Identify energy issues from power trace	
• **Identify execution issues** ◦ **Select dimensions** ◦ **Cluster** ◦ **Label outliers**	• *Identify background and no-sleep issues* ◦ *Dissimilarity analysis*

predictors to infer energy information at device, component, virtual machine or application level [21, 38, 40, 42, 65, 71, 82, 86]. Shuai et al. [30] and Ding et al. [44] propose approaches to obtaining energy information at source line level. The former requires the specific energy profile of the target systems. The latter utilizes advanced measurement techniques to obtain source line energy cost. Exceptionally, another line of research (e.g. [57], [53], [54] and [79]) attempts to detect energy issues relying on non-energy information. For example, Xiao et al. [57] analyse abnormal time-varying behaviours of apps to identify energy issues. But their works are not based-on the ground-truth of power measurement, thus the efficacy of their approach is unclear when applied in the wild.

Two pieces of work [35, 36] from Jabbarvand et al. are close to our work. Our work differs from theirs in three main aspects. First, their works mainly address the challenges of issue manifestation (how to trigger the issues), while our work addresses the challenges of both issue manifestation and detection (how to identify the real existence of the triggered issues). That is to say, their envisaged issues may not turn out to cause energy overuse. Second, they assume that the over-use of certain APIs is the main source of energy issues, which is also assumed by [17]. However, APIs cannot cover all usage of CPU, i.e. the main energy consumer [29] on smartphones. So their test cannot provoke most CPU-specific energy issues, a significant part of all energy issues. In contrast, our work assesses the CPU usage by tracing control-flow operations at code level. Third, their evaluations show the efficacy of their techniques for triggering previously-reported energy issues, but the efficacy for triggering previously-unknown issues is not justified due to the lack of issue-detection mechanisms.

It is worthy to notice that, the work of [35] also deals with special runtime context. But as we discussed above, their approach only involves how to trigger the issues hidden by special runtime context. They have no issue detection technology. So they use the previously-reported issues as subjects to evaluate the efficacy of their technique. Therefore, the efficacy for manifesting new issues are unevaluated. Furthermore, their work cannot handle CPU-specific energy issues, which accounts for 90.9% of all the issues caused by special runtime context, as seen in our experiment.

The work of Banerjee et al. [17] is the most relevant to ours. Table 7 lists the differences: 1) For preparation before testing, since their work [17] only takes I/O components into account, their technology is impracticable for the important CPU-bound apps (e.g. games) and CPU-related energy issues. They also neglected the special runtime contexts which can trigger 24.2% of energy issues. 2) W.r.t steering test direction, their work does not have this feature because they only consider two dimensions of testing space, i.e. I/O-related input sequences and `Normal` context, whereas our framework additionally tests five more dimensions including CPU-related and random input sequences, and three more runtime contexts. So our searching space for energy issues is exponentially enlarged; we thus designed practical online steering strategy to balance different kinds of inputs and contexts, and improve the issue-detection efficacy. 3) For identifying energy issues from power trace, as

we mentioned in Section 1, E/U theory [17] can hardly address execution issues (63.7% of all energy issues), by comparison, our framework employs advanced machine learning algorithm to analyse the energy use of apps and filter out these issues.

9.4 Fixing Energy Issues and Optimizing Software

A large amount of research effort on energy-saving for mobile devices has been focused on the main hardware components, such as the CPU, display and network interface. The CPU-related techniques involve dynamic voltage and frequency scaling [25, 73, 74], heterogeneous architecture [28, 46, 50] and computation offloading [41, 77]. Techniques targeting the display include dynamic frame rate tuning [33], dynamic resolution tuning [31, 51] and tone-mapping based back-light scaling [15, 34]. Network-related techniques try to exploit idle and deep sleep opportunities [52, 76], shape the traffic patterns [23, 69], trade-off energy against other design-criteria [20, 72, 85], and streaming with power-awareness [22, 32, 61]. Such work attempts to reduce energy costs by optimizing the hardware usage; there are also several pieces of work aiming at designing new hardware and devices [78, 80, 89]. Besides, another line of research work is dedicated to solving background and no-sleep issues [16, 18, 21, 55, 59].

Two pieces of work [19, 58] provide systematic approaches to optimizing software source code for energy-efficiency. In the former, Boddy et al. attempted to decrease the energy consumption of software by handling code as if it were genetic material so as to evolve to be more energy-efficient. In the latter, Irene et al. proposed a framework to optimize Java applications by iteratively searching for more energy-saving implementations in the design space.

10 CONCLUSION

In this paper, we conducted an empirical study on software energy issues in 27 well-maintained open-source mobile apps. Inspired by this study, we fully implemented a novel testing framework for detecting energy issues. It first statically analyses the source code of app subjects and then extracts the candidate input-sequences with large probability of causing energy issues. We also devised several artificial runtime contexts that can expose deeply-hidden energy issues. Our framework effectively examines apps with the inputs and contexts under a systematic scheme, and then automatically identifies energy issues from power traces. A large-scale experimental evaluation showed that our framework is capable of detecting a large number of energy issues, most of which existing techniques cannot handle. Finally, we showed how developers can utilize our test reports to fix the issues.

REFERENCES

[1] [n.d.]. *ADB.* developer.android.com/studio/profile/battery-historian.html.
[2] [n.d.]. *Battery monitor widget.* play.google.com/store/apps/details?id=ccc71.bmw.
[3] [n.d.]. *Better battery stats.* play.google.com/store/apps/details?id=com.asksven.betterbatterystats.
[4] [n.d.]. *GDB.* sourceware.org/gdb/onlinedocs/gdb/Backtrace.html.
[5] [n.d.]. *How Many Test Users in a Usability Study?* https://www.nngroup.com/articles/how-many-test-users/.
[6] [n.d.]. *Monsoon Power Monitor.* https://www.msoon.com/online-store.
[7] [n.d.]. *My battery drain analyze.* play.google.com/store/apps/details?id=com.WazaBe.android.BatteryDrains.
[8] [n.d.]. *Perfetto.* https://perfetto.dev.
[9] [n.d.]. *Telemetry.* www.chromium.org/developers/telemetry/run_locally.
[10] [n.d.]. *Trace view.* developer.android.com/studio/profile/traceview.html.
[11] [n.d.]. *Wake lock detector.* play.google.com/store/apps/details?id=com.uzumapps.wakelockdetector.noroot.
[12] [n.d.]. *ZDBox.* zdbox.en.uptodown.com/android/download.
[13] Sharad Agarwal, Ratul Mahajan, Alice Zheng, and Victor Bahl. 2010. Diagnosing Mobile Applications in the Wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (Monterey, California) *(Hotnets-IX)*. ACM, New York, NY, USA, Article 22, 6 pages. https://doi.org/10.1145/1868447.1868469
[14] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing,

NY, USA) *(SOSP '03)*. ACM, New York, NY, USA, 74–89. https://doi.org/10.1145/945445.945454

[15] Bhojan Anand, Karthik Thirugnanam, Jeena Sebastian, Pravein G. Kannan, Akhihebbal L. Ananda, Mun Choon Chan, and Rajesh Krishna Balan. 2011. Adaptive Display Power Management for Mobile Games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (Bethesda, Maryland, USA) *(MobiSys '11)*. ACM, New York, NY, USA, 57–70. https://doi.org/10.1145/1999995.2000002

[16] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. 2018. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Transactions on Software Engineering* 44, 5 (2018), 470–490. https://doi.org/10.1109/TSE.2017.2689012

[17] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 588–598. https://doi.org/10.1145/2635868.2635871

[18] A. Banerjee and A. Roychoudhury. 2016. Automated Re-factoring of Android Apps to Enhance Energy-Efficiency. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 139–150. https://doi.org/10.1109/MobileSoft.2016.038

[19] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing Energy Consumption Using Genetic Improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO '15)*. ACM, New York, NY, USA, 1327–1334. https://doi.org/10.1145/2739480.2754752

[20] Duc Hoang Bui, Yunxin Liu, Hyosu Kim, Insik Shin, and Feng Zhao. 2015. Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (Paris, France) *(MobiCom '15)*. ACM, New York, NY, USA, 14–26. https://doi.org/10.1145/2789168.2790103

[21] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (Paris, France) *(MobiCom '15)*. ACM, New York, NY, USA, 40–52. https://doi.org/10.1145/2789168.2790107

[22] X. Chen, T. Tan, G. Cao, and T. F. La Porta. 2020. Context-Aware and Energy-Aware Video Streaming on Smartphones. *IEEE Transactions on Mobile Computing* (2020), 1–1. https://doi.org/10.1109/TMC.2020.3019341

[23] C. Chiasserini and R.R. Rao. 2001. Improving battery performance by using traffic shaping techniques. *Selected Areas in Communications, IEEE Journal on* 19, 7 (Jul 2001), 1385–1394. https://doi.org/10.1109/49.932705

[24] L. Cruz, R. Abreu, J. Grundy, L. Li, and X. Xia. 2019. Do Energy-Oriented Changes Hinder Maintainability?. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 29–40. https://doi.org/10.1109/ICSME.2019.00013

[25] V. Devadas and H. Aydin. 2012. On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-Based Real-Time Embedded Applications. *IEEE Trans. Comput.* 61, 1 (Jan 2012), 31–44. https://doi.org/10.1109/TC.2010.248

[26] Mohammad N.t. Elbatta. 2012. *An improvement for DBSCAN algorithm for best results in varied densities*. Islamic University of Gaza, http://hdl.handle.net/20.500.12358/18778.

[27] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (Portland, Oregon) *(KDD'96)*. AAAI Press, 226–231. http://dl.acm.org/citation.cfm?id=3001460.3001507

[28] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. 2011. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *Micro, IEEE* 31, 2 (March 2011), 86–95. https://doi.org/10.1109/MM.2011.18

[29] M. Halpern, Y. Zhu, and V. J. Reddi. 2016. Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 64–76.

[30] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating Mobile Application Energy Consumption Using Program Analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 92–101. http://dl.acm.org/citation.cfm?id=2486788.2486801

[31] Songtao He, Yunxin Liu, and Hucheng Zhou. 2015. Optimizing Smartphone Power Consumption Through Dynamic Resolution Scaling. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (Paris, France) *(MobiCom '15)*. ACM, New York, NY, USA, 27–39. https://doi.org/10.1145/2789168.2790117

[32] W. Hu and G. Cao. 2015. Energy-aware video streaming on smartphones. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. 1185–1193. https://doi.org/10.1109/INFOCOM.2015.7218493

[33] Chanyou Hwang, Saumay Pushp, Changyoung Koh, Jungpil Yoon, Yunxin Liu, Seungpyo Choi, and Junehwa Song. 2017. RAVEN: Perception-aware Optimization of Power Consumption for Mobile Games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah, USA) *(MobiCom '17)*. ACM, New York, NY, USA, 422–434. https://doi.org/10.1145/3117811.3117841

[34] Ali Iranli and Massoud Pedram. 2005. DTM: Dynamic Tone Mapping for Backlight Scaling. In *Proceedings of the 42Nd Annual Design Automation Conference* (Anaheim, California, USA) *(DAC '05)*. ACM, New York, NY, USA, 612–617. https://doi.org/10.1145/1065579.

1065741
[35] Reyhaneh Jabbarvand, Jun-Wei Lin, and Sam Malek. 2019. Search-Based Energy Testing of Android. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1119–1130. https://doi.org/10.1109/ICSE.2019.00115
[36] Reyhaneh Jabbarvand and Sam Malek. 2017. MDroid: An Energy-Aware Mutation Testing Framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 208–219. https://doi.org/10.1145/3106237.3106244
[37] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075
[38] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. 2010. Virtual Machine Power Metering and Provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1807128.1807136
[39] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. 2008. Dustminer: Troubleshooting Interactive Complexity Bugs in Sensor Networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems* (Raleigh, NC, USA) *(SenSys '08)*. ACM, New York, NY, USA, 99–112. https://doi.org/10.1145/1460412.1460423
[40] J. Koo, K. Lee, W. Lee, Y. Park, and S. Choi. 2016. BattTracker: Enabling energy awareness for smartphone using Li-ion battery characteristics. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524422
[41] K. Kumar and Y. Lu. 2010. Cloud Computing for Mobile Users: Can Offloading Computation Save Energy? *Computer* 43, 4 (April 2010), 51–56. https://doi.org/10.1109/MC.2010.98
[42] Seokjun Lee, Chanmin Yoon, and Hojung Cha. 2014. User Interaction-based Profiling System for Android Application Tuning. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Seattle, Washington) *(UbiComp '14)*. ACM, New York, NY, USA, 289–299. https://doi.org/10.1145/2632048.2636091
[43] Ding Li and William G. J. Halfond. 2014. An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software* (Hyderabad, India) *(GREENS 2014)*. ACM, New York, NY, USA, 46–53. https://doi.org/10.1145/2593743.2593750
[44] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. 2013. Calculating Source Line Level Energy Information for Android Applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. ACM, New York, NY, USA, 78–89. https://doi.org/10.1145/2483760.2483780
[45] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 153–163. https://doi.org/10.1145/3213846.3213857
[46] Xianfeng Li, Guikang Chen, and Wen Wen. 2017. Energy-Efficient Execution for Repetitive App Usages on Big.LITTLE Architectures. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. ACM, New York, NY, USA, Article 44, 6 pages. https://doi.org/10.1145/3061639.3062239
[47] X. Li and J. P. Gallagher. 2016. A Source-Level Energy Optimization Framework for Mobile Applications. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 31–40. https://doi.org/10.1109/SCAM.2016.12
[48] Xueliang Li, Yuming Yang, Yepang Liu, John P. Gallagher, and Kaishun Wu. 2020. Detecting and Diagnosing Energy Issues for Mobile Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Los Angeles, CA, USA) *(ISSTA 2020)*. ACM, New York, NY, USA. https://doi.org/10.1145/3395363.3397350
[49] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. ACM, New York, NY, USA, 141–154. https://doi.org/10.1145/781131.781148
[50] Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. 2012. Reflex: Using Low-power Processors in Smartphones Without Knowing Them. *SIGPLAN Not.* 47, 4 (March 2012), 13–24. https://doi.org/10.1145/2248487.2150979
[51] H. Lin, C. Hung, P. Hsiu, and T. Kuo. 2018. Duet: An OLED GPU Co-management Scheme for Dynamic Resolution Adaptation. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465887
[52] Jiayang Liu and Lin Zhong. 2008. Micro Power Management of Active 802.11 Interfaces. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (Breckenridge, CO, USA) *(MobiSys '08)*. ACM, New York, NY, USA, 146–159. https://doi.org/10.1145/1378600.1378617
[53] Yi Liu, Jue Wang, Chang Xu, and Xiaoxing Ma. 2017. NavyDroid: Detecting Energy Inefficiency Problems for Smartphone Applications. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware* (Shanghai, China) *(Internetware'17)*. Association for Computing Machinery, New York, NY, USA, Article 8, 10 pages. https://doi.org/10.1145/3131704.3131705
[54] Yepang Liu, Chang Xu, S.C. Cheung, and Jian Lv. 2014. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering* 40, 9 (Sept 2014), 911–940. https://doi.org/10.1109/TSE.2014.2323982

[55] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and Detecting Wake Lock Misuses for Android Applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2016*.

[56] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[57] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul, and Geoffrey M. Voelker. 2013. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 57–70. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ma

[58] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 503–514. https://doi.org/10.1145/2568225.2568297

[59] Marcelo Martins, Justin Cappos, and Rodrigo Fonseca. 2015. Selectively Taming Background Android Apps to Improve Battery Lifetime. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 563–575. https://www.usenix.org/conference/atc15/technical-session/presentation/martins

[60] A. Memon, I. Banerjee, and A. Nagarajan. 2003. GUI ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* 260–269. https://doi.org/10.1109/WCRE.2003.1287256

[61] Jiayi Meng, Qiang Xu, and Y. Charlie Hu. 2021. Proactive Energy-Aware Adaptive Video Streaming on Mobile Devices. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 303–316. https://www.usenix.org/conference/atc21/presentation/meng

[62] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, Reporting, and Fixing Performance Bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 237–246. http://dl.acm.org/citation.cfm?id=2487085.2487134

[63] Adrian Nistor and Lenin Ravindranath. 2014. SunCat: Helping Developers Understand and Predict Performance Problems in Smartphone Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. ACM, New York, NY, USA, 282–292. https://doi.org/10.1145/2610384.2610410

[64] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2011. Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (Cambridge, Massachusetts) *(HotNets-X)*. ACM, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2070562.2070567

[65] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2012. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/2168836.2168841

[66] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. 2012. What is Keeping My Phone Awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (Low Wood Bay, Lake District, UK) *(MobiSys '12)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/2307636.2307661

[67] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. 2014. Integrated CPU-GPU Power Management for 3D Mobile Games. In *Proceedings of the 51st Annual Design Automation Conference on*. 1–6.

[68] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. 2000. Empirical studies of software engineering: a roadmap. In *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*, Anthony Finkelstein (Ed.). ACM, 345–355. https://doi.org/10.1145/336512.336586

[69] C. Poellabauer and K. Schwan. 2004. Energy-aware traffic shaping for wireless real-time applications. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*. 48–55. https://doi.org/10.1109/RTTAS.2004.1317248

[70] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 176–186. https://doi.org/10.1145/3213846.3213873

[71] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 107–120. http://dl.acm.org/citation.cfm?id=2387880.2387891

[72] A. Sehati and M. Ghaderi. 2017. Energy-delay tradeoff for request bundling on smartphones. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057063

[73] A. Shye, B. Ozisikyilmaz, A. Mallik, G. Memik, P. A. Dinda, R. P. Dick, and A. N. Choudhary. 2008. Learning and Leveraging the Relationship between Architecture-Level Measurements and Individual User Satisfaction. In *2008 International Symposium on Computer Architecture*. 427–438. https://doi.org/10.1109/ISCA.2008.29

[74] A. Shye, Y. Pan, B. Scholbrock, J. S. Miller, G. Memik, P. A. Dinda, and R. P. Dick. 2008. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 188–199. https://doi.org/10.1109/MICRO.2008.4771790

[75] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. 2009. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. ACM, New York, NY, USA, 168–178. https://doi.org/10.1145/1669112.1669135

[76] Jacob Sorber, Nilanjan Banerjee, Mark D. Corner, and Sami Rollins. 2005. Turducken: Hierarchical Power Management for Mobile Devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services* (Seattle, Washington) *(MobiSys '05)*. ACM, New York, NY, USA, 261–274. https://doi.org/10.1145/1067170.1067198

[77] K. Sucipto, D. Chatzopoulos, S. Kosta±, and P. Hui. 2017. Keep your nice friends close, but your rich friends closer — Computation offloading using NFC. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057147

[78] Tim Tuan, Sean Kao, Arif Rahman, Satyaki Das, and Steve Trimberger. 2006. A 90Nm Low-power FPGA for Battery-powered Applications. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '06)*. ACM, New York, NY, USA, 3–11. https://doi.org/10.1145/1117201.1117203

[79] Jue Wang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. E-GreenDroid: Effective Energy Inefficiency Analysis for Android Applications. In *Proceedings of the 8th Asia-Pacific Symposium on Internetware* (Beijing, China) *(Internetware '16)*. Association for Computing Machinery, New York, NY, USA, 71–80. https://doi.org/10.1145/2993717.2993720

[80] Li Wang, Matthew French, Azadeh Davoodi, and Deepak Agarwal. 2006. FPGA Dynamic Power Minimization Through Placement and Routing Constraints. *EURASIP J. Embedded Syst.* 2006, 1 (Jan. 2006), 7–7. https://doi.org/10.1155/ES/2006/31605

[81] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1970. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Selected tables in mathematical statistics* 1 (1970), 171–259.

[82] Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. 2013. V-edge: Fast Self-constructive Power Modeling of Smartphones Based on Battery Voltage Dynamics. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 43–55. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/xu_fengyuan

[83] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding Low-Utility Data Structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/1806596.1806617

[84] Hideo Yamamoto, Tomohiro Hirano, Kohei Muto, Hiroki Mikami, Takashi Goto, Dominic Hillenbrand, Moriyuki Takamura, Keiji Kimura, and Hironori Kasahara. 2014. OSCAR Compiler Controlled Multicore Power Reduction on Android Platform. In *Languages and Compilers for Parallel Computing*. Springer International Publishing, Cham, 155–168.

[85] Zhisheng Yan and Chang Wen Chen. 2016. RnB: Rate and Brightness Adaptation for Rate-distortion-energy Tradeoff in HTTP Adaptive Streaming over Mobile Devices. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking* (New York City, New York) *(MobiCom '16)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/2973750.2973780

[86] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. 2012. AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 36–36. http://dl.acm.org/citation.cfm?id=2342821.2342857

[87] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (Zurich, Switzerland) *(MSR '12)*. IEEE Press, Piscataway, NJ, USA, 199–208. http://dl.acm.org/citation.cfm?id=2664446.2664477

[88] Lide Zhang, Mark S. Gordon, Robert P. Dick, Z. Morley Mao, Peter Dinda, and Lei Yang. 2012. ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (Tampere, Finland) *(CODES+ISSS '12)*. Association for Computing Machinery, New York, NY, USA, 363–372. https://doi.org/10.1145/2380445.2380503

[89] Lin Zhong and Niraj K. Jha. 2005. Energy Efficiency of Handheld Computer Interfaces: Limits, Characterization and Practice. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services* (Seattle, Washington) *(MobiSys '05)*. ACM, New York, NY, USA, 247–260. https://doi.org/10.1145/1067170.1067197

[90] Y. Zhu, M. Halpern, and V. J. Reddi. 2015. Event-based scheduling for energy-efficient QoS (eQoS) in mobile Web applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 137–149. https://doi.org/10.1109/HPCA.2015.7056028