An Empirical Study on the Impact of Deep Parameters on Mobile App Energy Usage

***Abstract*—Improving software performance through configuration parameter tuning is a common activity during software maintenance. Beyond traditional performance metrics like latency, mobile app developers are interested in reducing app energy usage. Some mobile apps have centralized locations for parameter tuning, similar to databases and operating systems, but it is common for mobile apps to have hundreds of parameters scattered around the source code. The correlation between these "deep" parameters and app energy usage is unclear. Researchers have studied the energy effects of deep parameters in specific modules, but we lack a systematic understanding of the energy impact of mobile deep parameters.**

In this paper we empirically investigate this topic, combining a developer survey with systematic energy measurements. Our motivational survey of 25 Android developers suggests that developers do not understand, and largely ignore, the energy impact of deep parameters. To assess the potential implications of this practice, we propose a deep parameter energy profiling framework that can analyze the energy effect of deep parameters in an app. Our framework identifies deep parameters, mutates them based on our parameter value selection scheme, and performs reliable energy impact analysis. Applying the framework to 16 popular Android apps, we discovered that deep parameter-induced energy inefficiency is rare. We found only 2 examples out of 1644 deep parameters for which a different value would significantly improve its app's energy usage. A detailed analysis found that most deep parameters have either no energy impact, limited energy impact, or an energy impact only under extreme values. Our study suggests that it is generally safe for developers to ignore the energy impact when choosing deep parameter values in mobile apps.

I. Introduction

Improving energy efficiency is one of a mobile app developer's software maintenance activities. App users desire efficient energy usage [1], and the resulting improvement in accessibility can benefit individuals and societies [2], [3]. Mobile platform vendors, *e.g.,* Google and Apple, also advise app developers to optimize app energy usage [4], [5].

One potential strategy to reduce a mobile app's energy usage is to tune its configuration parameters. All software includes configuration parameters to help it be adapted to different contexts. Mobile apps are no exception: in addition to the parameters explicitly exposed in resource files and other configuration files, these apps have many *deep parameters*, *i.e.,* parameters that are scattered around the source code to control runtime behaviors including buffer sizes, task frequencies, and UI layout positions. Researchers have shown that tuning parameters can improve the performance of conventional software [6]–[8]. However, mobile deep parameters are often overlooked by developers, and little is known about the energy impacts of deep parameters in mobile apps. Prior works have only studied the energy impacts of deep parameters in specific modules [9] or libraries [10], [11], not systematically.

We investigated the energy impact of mobile deep parameters using mixed methods [12], combining a developer survey with systematic energy measurements. In our survey of 25 Android app developers, we found that developers are uncertain about the energy impact of deep parameters and do not usually consider energy when choosing parameter values.

To measure the implications of developers' practices on deep parameters, we propose a parameter-centric energy profiling framework. The framework extracts deep parameters from the app, mutates them based on our parameter value selection scheme, and measures the changes in energy drain. Our framework overcomes several challenges: identifying deep parameters, choosing appropriate mutation values, and reliably measuring the energy impact.

We systematically measured deep parameter energy effects in 16 popular open-source Android apps. Among the 1644 parameters tested, only 2 are set to poor values. Further analysis shows that the rest of the parameters either have no energy effect, have limited energy effects, or only have energy effects under extreme values that developers can typically avoid based on their domain knowledge. We conclude that it is generally safe for developers to ignore energy effects when choosing deep parameter values — developers must look elsewhere for energy-reducing refactorings.

Our study makes the following contributions:

- We describe the practices of mobile app developers on deep parameters and energy optimization (N=25) (§IV).
- We propose a framework for parameter-centric profiling in Android, automatically identifying deep parameters and measuring their energy impacts (§V).
- We perform the first systematic study of the energy impact of deep parameters in Android apps (N=16) (§VII). We describe the roles of deep parameters in these apps, and identify three energy categories of deep parameters.
- We open source our framework [1] and full survey and experiment results [**?**] to enable further exploration from the research community.

II. Background and Definitions

A. Configuration Tuning in General

Many categories of software can be configured for different usage scenarios and deployment environments. Such software includes databases, stream processing frameworks, web

[1] https://github.com/qiangxu1996/literal-mutator

```
1  Bitmap.createBitmap(320, 240, ARGB_8888);
2  byte[] serverVersion = new byte[512];
3  sock.setSocketTimeout(0);
4  layoutParams.width = 12;
```

Fig. 1: Real Android code snippets; deep parameters in red.

servers, codecs, and others. Their configuration parameters are typically exposed through configuration files, command-line interfaces, or certain data structures [6]. For example, to configure the video codec `x265`, one can pass command-line arguments to the executable [13], or specify the `x265_param` data structure through its API [14].

In addition to their implications on software functionality, configuration parameters may also impact the performance. For example, by tuning its ∼200 configuration parameters, MySQL database throughput can be improved by 6x and its latency reduced by 3x on common benchmarks [8]. As configuration tuning is an NP-hard problem [15], configuration tuners aim to efficiently search the configuration space and recommend configuration values for a given workload (*e.g.,* [6], [7]). In these contexts, auto-tuners are able to focus on the performance optimization task because the software parameters are clearly defined (*e.g.,* in configuration files).

B. Parameters in Mobile Apps

Mobile apps also contain configurable parameters that control various aspects of the apps. As in most user-facing software, latency is a major performance metric in mobile apps. However, *energy* is also a key metric in mobile apps. Similar to configuration tuning for other performance metrics, other researchers have provided preliminary evidence that some parameters impact app energy consumption. Canino *et al.* [9] showed that GPS configuration parameters can be tuned to meet specified energy consumption SLAs. Similarly, Bokhari *et al.* [10], [11] optimized the energy consumption of the Rebound physics library by tuning the integer and double parameters in it. However, it is unclear whether such energy-affecting parameters are common in general Android apps.

This problem is challenging because, in contrast to the software discussed above, parameters in Android apps are more often scattered all around the source code rather than stored at central places. We speculate that parameter centralization occurs when the software is used by skilled operators such as database administrators. Most mobile apps are designed for unskilled users, and so there is little customer demand to centralize and expose parameters. This challenge must be overcome in order to understand the impact of parameters on mobile app energy usage.

C. Definitions

Following Bokhari *et al.* [10], [11], we define a **deep configuration parameter** as *a constant in app source code that can be changed by app developers, but does not affect app functionality*. In other words, all app components should function properly when tuning a deep parameter's value, with "minimal" impact on user experience. This property can be determined by examining the source code or the runtime app behavior.[2] Some examples are given in Figure 1, *e.g.,* buffer sizes, timeouts, and UI element sizes, and the parameters can be numeric, Boolean, or enumerator references.

III. Research Questions

We seek to understand the energy impact of Android deep parameters. Developers' awareness of the energy impact of deep parameters and their strategy in deciding parameter values may impact app energy efficiency. We also want to measure and understand the energy effects of tuning deep parameters. Operationalized, our research questions are:

RQ1: What are the energy impacts of parameters in developers' eyes?
RQ2: How do developers choose parameter values?
RQ3: Is deep parameter-induced energy inefficiency common among apps?
RQ4: When and why do (and do not) deep parameters impact app energy consumption?

We investigated RQ1-2 with a developer survey. We studied RQ3-4 through a systematic energy measuring experiment.

IV. Developer Perspectives

We surveyed Android developers to better understand their perceptions and practices regarding Android parameters.

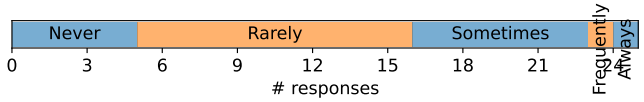
A. Methodology

We designed an IRB-approved survey to obtain mobile app developers' perspectives on RQ1 and RQ2. The RQs were operationalized into 6 demographic questions and 13 study-specific questions. These questions included closed- and open-ended questions across three topics: (1) the nature of the parameters in their apps; (2) their perceptions of parameters' energy impacts; and (3) their process when choosing parameter values. Rather than using interviews to elicit relevant topics for the survey, we based the questions on (a) our own expertise from studying and developing energy-efficient mobile apps; and (b) preliminary findings and observations from our energy experiments (detailed in sections V to VII). To improve instrument validity and reduce bias, we followed best practices during survey design [16], [17], *e.g.,* avoiding leading questions, and refined the survey through two rounds of pilot studies with graduate students.

To ensure participant's understanding of the term "parameter", we provided the following definition before survey questions: "*Android apps contain parameters that control various aspects of the apps. Common types of parameters include upper/lower bounds, UI layout sizes/positions, buffer-/cache sizes, thread counts, timeouts, task frequencies, etc. They could be hard-coded numbers, constants, or dynamically varying. In this survey, we are interested in the parameters*

[2] A high-quality test suite would be a suitable oracle, but we found these suites inadequate in our experiments.

(a) For what proportion of the parameters in your app are you confident about the energy impact of changing them?



(b) How often do you consider energy consumption while choosing parameter values?

Fig. 2: Distribution of responses to developer perception of parameters' energy impact and how they pick parameter values

that are accessible to developers, i.e., not in-app settings."[3] Respondents were asked to answer in terms of the app they spent the most time developing.

We distributed the survey to Android developers through multiple channels. We posted the survey on popular forums frequented by Android developers (subreddits `r/androiddev` and `r/mAndroidDev`) and developers in general (Hacker News and DEV), and social media groups of Android developers (LinkedIn, Facebook, and Slack). We also contacted Android developers in our professional networks. Survey participants were not compensated.

We received 25 non-blank responses: 15 from forums and social media groups, and 10 from professional connections.

B. Results

a) Demographics: The median respondent has 6-10 years of software development experience and 3-5 years of Android development experience. Respondents work on apps from 13⁄37 categories defined by Google Play [18]. For 21⁄25 responses, the answers describe commercial app development.

b) RQ1: Developer Perception of Energy Impact: Developers are concerned about app energy usage: 10 of the 25 respondents monitor energy consumption. However, most of these respondents use coarse-grained measurements like CPU usage or battery statistics. These tools can detect severe or specific types of energy bugs (*e.g.,* wake lock [19]), but are difficult to use for energy tuning. Perhaps in consequence, only 3 respondents are confident about the energy impacts of "a lot" of their apps' parameters (Figure 2a).

> **Finding 1**: Around half of mobile app developers measure app energy usage. Few developers (12%) are confident about the energy impact of parameters.

c) RQ2: Picking Parameter Values: Our respondents said that when they choose parameter values, their top concerns are app functionality and user experience. Only 2 of the 25 developers frequently or always consider energy consumption when parameterizing (Figure 2b). The reason might again lie in the fact that developers don't have handy tools for energy tuning. As developers have only limited confidence in parameters' energy impacts, further experiments are still needed to validate developers' choices.

> **Finding 2**: Only 8% of developers frequently consider energy consumption when choosing parameter values.

d) Parameter Locations: Other data from our survey informed our parameter measurement approach. For parameters in source files, our respondents estimated that the majority are scattered across the codebase; only a third of respondents described their apps as having substantial parameter centralization in files like `Config.java` or `Constants.java`. This finding is consistent with our observations of open-source Android apps, discussed in §II-B.

V. Deep Parameter Testing Framework

The developers in our survey indicated that they rarely consider energy consumption when picking deep parameter values. This practice does not necessarily mean that they make poor choices. Developers might intuitively make energy-efficient choices; energy-efficient choices may correlate with choices that improve usability; or deep parameters may not have a substantial effect on energy usage. However, since developers told us that they do not consider the energy effects of parameters, our governing hypothesis is that *ignoring energy effect results in suboptimal deep parameterizations*. To test the hypothesis, we propose a framework that mutates every deep parameter and checks if the change reduces energy usage.

An overview of the mutate-and-test process is shown in Figure 3. As deep parameters are scattered in the source code, we first extract deep parameters from the set of all constants. For each of the parameters extracted, we try several new values based on our parameter mutation scheme, and measure the energy consumption of the app variants. We manually validate all parameterizations that reduce energy use, and finally report any discovered energy-reducing parameters.

We present the details of each step below. Many design details are informed by preliminary experiments and observations. The primary design constraint is *time*: testing one parameter takes a long time (on average 22 minutes), as we need to perform repeated tests for multiple parameter values on the phone; and these tests are not easily scaled, since we need a physical device to get accurate energy measurements.

We describe our experiment applying the framework to 16 popular Android apps in §VI and the results in §VII.

A. Deep Parameter Extraction

Deep parameters are scattered around source files. Thus, to test the parameters, one design option is to test all constants, regardless they are deep parameters or not. However, app typically has thousands of constants. Testing all of them is prohibitively time-consuming. We therefore use a mix of manual and automatic filtering to distinguish deep parameters from other constants.

[3]This definition includes both deep (cf. §II-C) and traditional parameters. Our results thus give a broader perspective on the energy tuning practices of mobile app developers. As deep parameters are a subset of this definition, our survey results also characterize engineering practices for deep parameters.

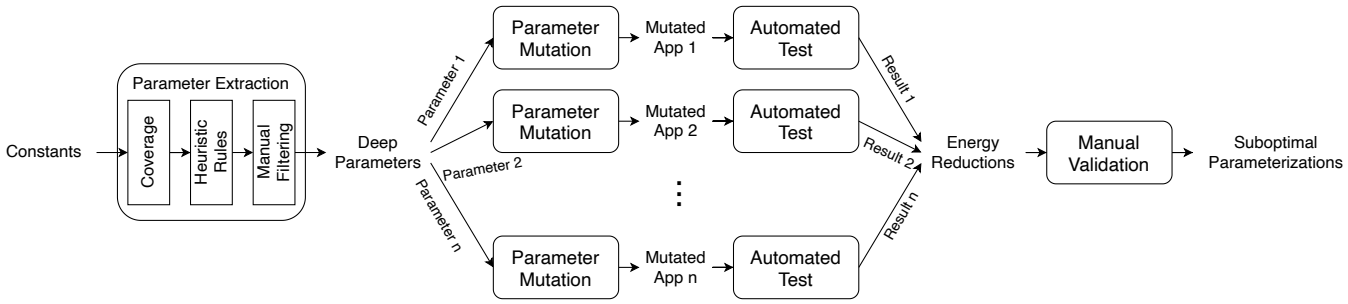


Fig. 3: Workflow of the mutate-and-test process.

a) Usage Scenarios and Parameter Coverage: Our framework tests an app using UI automation scripts over several of its usage scenarios. After we identify these usage scenarios, we use code coverage to identify the candidate deep parameters. A usage scenario may target a particular feature set (*e.g.,* features to be used in a low-power mode), and only some deep parameters will affect the energy usage in this scenario. Deep parameters that are not used cannot have an energy effect. Based on this insight, we record the line-level code statement coverage and filter out parameters that are not covered. This filtering is performed prior to the energy measurements, thus energy measurements are conducted on a non-instrumented app and do not have instrumentation effects.

b) Heuristic Rules: We noticed that true deep parameters are more likely to occur in certain code constructs, while constants in some other code constructs are unlikely to be deep parameters. For example, variable initializers and method call arguments tend to be deep parameters; bitwise operator arguments typically not.

Using "negative patterns" matching non-parameter constants minimize the chance that we incorrectly filter out deep parameters. Thus, we assemble a set of patterns that are shared by portions of the non-parameter constants, and filter out constants that match any pattern. The patterns are extracted by manually classifying constants in a file to be deep parameters or not, and identifying any patterns that the non-parameter constants may share.

After manually inspecting the files with the most constants, we derived heuristic rules as shown in Table I. Each rule may be applicable to one or several parameter types. As the rules are more pertinent to programming language idioms [20] than an individual developer's habits, they apply across apps. However, as the rules depend on common development practices, they may incorrectly filter parameters (false negatives).

c) Manual Filtering: The preceding filters are automated. However, many constants remain for consideration. We manually examine the remaining constants and filter out the non-parameter constants based on their semantic. For example, error codes are not parameters, since mutating them will result in the app reporting the wrong kind of error, and thus affecting app functionality.

B. Deep Parameter Mutation

TABLE I: Heuristic rules for filtering non-parameter constants.

Type	Rule	Example
Num	Array index	a[0]
Num	Comparison with 0 or 1	a.size() > 0
Num	Plus 1 or 2 or minus 1	a.length() - 1
Num	For loop initialization	for (int i = 0; ...)
Num	Ignored methods	s.substring(0, 4)
Bool	One argument method call	item.setVisible(true)
Enum	Time unit	convert(5, DAYS)
Enum	Locale	toLowerCase(US, str)
All	Condition	if (a.size() > 0)
All	Return value	return 0
All	Multiple writes to variable	See Figure 4

```
class Foo {
  int counter = 0;
  void count() { counter++; }
  void reset() { counter = 0; }
}
```

Fig. 4: Example of multiple writes. `counter` is updated in multiple places, so the `0` constants are not considered deep parameters.

We mutate and test each deep parameter in isolation. It is intractable to test all possible values for each parameter, thus sampling is used for value selection. We choose new values carefully to maximize the chance that we observe a parameter's energy impact. For example, the energy consumption may not change if we choose values too close to the original one, yet we may crash the app if we choose values too far away. On other hand, the new values do not need to be optimal, as we can perform further investigation as long as we can observe an energy reduction.

Based on preliminary experiments, we developed a guideline for choosing new values for numeric parameters:

- Choose values from both sides of the original value; most parameters are monotonic in terms of their energy effects.
- A factor of 8 will expose the difference (if any) in energy consumption, unless the parameter only has energy impact under extreme cases (cf. §VII-C3).

We further fine-tune the values chosen based on the original parameter value, as we observed that the original value indicates the valid value range to some extent. For example, 0 is invalid for many positive integer parameters, while floating-

TABLE II: Mutated values for numeric parameters.

Original value	New values
0	0xffffff, 255, 8
1	8, 0
x (> 1, int)	$x * 8$, $\max(x/8, 1)$
0.0	0.5, 1.0
x ($0 < x < 1$, float)	$1 - (1 - x)/8$, $x/8$
x (≥ 1, float)	$x * 8$, $x/8$

```
while (!onScreen(item))
  swipe();
```

Fig. 5: UI automation code with potentially non-deterministic test interactions. This could lead to flaky test results.

point parameters with original values between 0 and 1 are highly likely to be valid only between 0 and 1.

Combining the guidelines and the fine-tuning, we build a parameter mutation scheme for numeric parameters as shown in Table II. Integer parameters with original value 0 have versatile semantics and do not fit into our guidelines. Thus, we choose several special values commonly used in programming to maximize the chance that some values suit the semantics.

Boolean and enumerator parameters are simpler than numeric parameters. We invert Boolean parameters, and for enumerations we randomly choose three additional values.

C. Automated Testing

The high-level workflow of automated testing is simple: we drive each app with a deterministic UI automation script. For each deep parameter, we measure the energy drain for both the unmodified app and the app with a new parameter value, and finally compare the results to see if the parameter can reduce the app energy drain. However, to ensure that the tests are reproducible, statistically solid, having minimal false positives and false negatives, and faithfully reflect the effects of the parameters, every step needs to be carefully thought out.

1) UI Automation Script: There has been a large body of research on automated UI testing for Android apps [21]–[26]. However, these works aim at improved code coverage while our automated testing instead focuses on reproducibility.

We design one UI automation script for each test scenario. While it is relatively easy to write a script that runs for a couple of times, extra caution is needed to design a script that runs thousands of times and ensures everything is reproducible at the same time. We enumerate four lessons we learned from our experience.

a) Ensure the interactions are deterministic: Executing the same script each time does not guarantee that the app performs the same actions. For example, the problematic code in Figure 5 swipes on a scrollable list until it finds the desired item. If the list contents or order vary, the app behavior will also vary, affecting energy usage. Avoid such loops with non-deterministic terminating conditions.

b) Ensure the test data are also deterministic: App behaviors depend on both the interactions and the data fed into the apps. Watching different videos or reading different posts may consume different amounts of energy. This issue is easy to solve for local apps like galleries or file managers, but harder for apps that rely on remote content. For example, the most popular posts on Reddit change. We address this by accessing static content whenever possible. In the case of the Reddit client `slide` (cf. Table III), instead of fetching trending posts, we fetch the most popular posts of all time.

c) Save app data to save bandwidth and time: Many apps need to download a large amount of data when opened for the first time. For example, `fdroid` needs to download tens of megabytes of metadata for all apps in the store. While this is acceptable for a small test, downloading the data several hundred times can easily lead to protective measures like reduced bandwidth or even blockage on the server-side.

On the other hand, preparing an app for a test may be very time-consuming. For example, to realistically test a password manager app, the app's database should have dozens of password entries. However, popping the database with so many entries for every fresh install takes time.

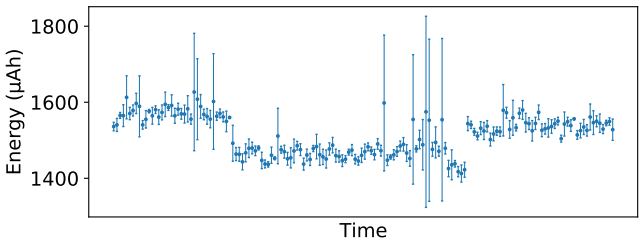
The solution for both scenarios is to utilize the data export functionality provided by many apps. By exporting the bootstrap data, all subsequent tests only need to import them locally after app installation.

d) Pay attention to the server state: For many apps requiring account login, part of the state is stored on the server-side. Without resetting the server state, the app may behave non-deterministically. For example, the server of the instant messenger `conv` recognizes the phone as a new device every time the app reinstalls. In the end, the server maintains thousands of "devices", which drastically changes the app behavior. As servers are black boxes, this type of problem is hard to diagnose. Our solutions were app-specific.

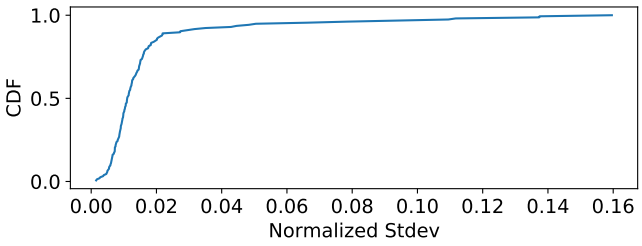
2) Back-to-Back Testing: Due to the time-dependent network condition and server load, the app energy consumption is changing over time for many apps with network access. Figure 6a shows the energy consumption of `ap` over a period of 3 days. While adjacent data points typically have similar energy consumption, the energy consumptions of distant data points can differ by as much as 14%. To reduce the influence of such time-dependent energy consumption drifts, we rerun the unmodified app before testing each deep parameter, and only compare the adjacent tests.

3) Hypothesis Testing: To determine whether a parameter value reduces energy consumption, we run both the unmodified app and the modified app 5 times (denoted as $B_1, \ldots, B_5$ and $P_1, \ldots, P_5$ respectively), and perform Student's t-test with the null hypothesis being mean(P_i) = mean(B_i) and the alternative hypothesis being mean(P_i) < mean(B_i), and a significance level of 0.05.

However, when applying the above hypothesis testing, we noticed that many energy drain reductions are due to small energy consumption fluctuations, rather than the parameters. To reduce such false positives, we set a threshold t_d for each



(a) Energy consumption over a period of 3 days. The error bar represents the standard deviation of the 5 runs.



(b) The CDF of the normalized standard deviations in Figure 7a.

Fig. 6: Energy consumption of the unmodified `ap` app. Each data point represents 5 runs.

app and filter out energy differences smaller than the threshold. Specifically, instead of performing the t-test for P_i and B_i, we now perform t-test for P_i and B'_i, where $B'_i = (1 - t_d)B_i$. Since P_i and B'_i have different variances, we switch from Student's t-test to Welch's t-test.

4) Stability Threshold: In the last section, we addressed the false positives caused by energy fluctuations. However, energy fluctuations can also introduce false negatives. Figure 6b plots the CDF of the normalized standard deviations (σ/μ) using the same data as Figure 6a. Although the standard deviation is low ($< 3\%$) in most cases, intermittently it can reach 16% of the mean. The chance of passing the t-test is minimal with such a high standard deviation. In such cases, we choose another small threshold t_s for each app, and discard results until the normalized standard deviation is back to normal (less than t_s). Both t_d and t_s are determined through experiments (cf. §VI).

D. Manual Validation

In the last step, we manually validate the energy-reducing cases to make sure that they are really caused by the parameter value changes, and the app functionalities are not impacted. Sometimes the energy fluctuation is too large and abrupt to be filtered out by the t-test threshold, while in other cases the app may not function normally with the parameter change.

E. Alternative Design

An alternative design for estimating the energy impact of deep parameters utilizes static or dynamic analysis. For each deep parameter, we can identify the dependent code segments by performing static data and control dependence analysis [27] or dynamic taint analysis [28], [29]. During execution, the energy consumption of the dependent code segments is measured and attributed to the corresponding parameter. Such an approach has the advantage of measuring the energy impacts of all deep parameters at once, but it also faces a number of challenges.

First, the dependencies between a deep parameter and the relevant code segments are often hard to track, or require *ad hoc* customizations to achieve good coverage. For example, we observed that many dependencies span programming language boundaries, or involve inter-process or inter-device (*e.g.,* client-server) communication.

Second, it is also hard to determine the right granularity of the code segments for dependence analysis. Dependence analysis at the branch level of branches has the advantage that the causal relationship between parameter values and branch conditions is easy to analyze. However, we observed that many deep parameters affect app energy consumption in ways other than controlling branch conditions. For example, app energy consumption can be affected by controlling the timer duration or thread count. Alternatively, we can perform the analysis at the method granularity by tracking the dependency between deep parameters and method call arguments. However, the relation between the parameter value and the method energy consumption is often opaque, if there is any relation at all.

In view of these challenges, we chose to apply the parameter mutation approach detailed earlier and leave improvements on program analysis tools for future work.

F. Implementation

We implemented our framework in 3.5 KLoC: parameter analysis and mutation (in Spoon [30]); UI automation (Appium [31]); and coverage analysis (JaCoCo [32]). App source code is required since we perform parameter analysis by examining the source code syntax tree.

The app needs to be rebuilt for each parameter mutation. Building apps and running test scripts are done in parallel, so that both the desktop and the phone can be fully utilized. We also make our test framework fully reentrant, and thus different tests can run on different phones independently.

VI. Experimental Design

We perform our experiments using 16 popular open-source Android apps. We choose the apps from 16 different categories to make sure our findings are not restricted to certain app categories. As we use Spoon to analyze app source code, we restrict ourselves to apps mostly (>70%) written in Java. Table III summarizes each app.

We design one test scenario for each app based on their typical usages (Table III). The lengths of the test cases are between 30 and 60 seconds.

The stability threshold t_s is determined by first running the experiments without the threshold. We then draw the CDF of the normalized standard deviations as in Figure 6b, and choose the turning point of the CDF curve as t_s. To determine the other threshold t_d, we rerun the parameters that pass the standard t-test ($t_d = 0$), and do the t-test again on the new measurements. Those that only pass the first t-test are considered due to energy fluctuations instead of the parameters themselves. Then we choose the minimum t_d that filters out the fluctuating ones while keeping the rest.

TABLE III: Tested apps and their test configurations. The popularity statistic (installs) is from Google Play.

App (Abbr.)	Category	Installs	Version	Test Scenario	t_s	t_d
SAI (sai)	App installer	5M+	4.5	Install 2 apps	0.02	0.01
ConnectBot (cb)	SSH client	4M+	1.9.7	View 6 Python files using vi	0.03	0.03
AnySoftKeyboard (ask)	Keyboard	2M+	1.10-rc4	Type username and password	0.03	0.01
KeePassDroid (kpd)	Password manager	2M+	2.5.12	Copy 8 password entries	0.03	0.03
Amaze File Manager (amaze)	File manager	1M+	3.4.3	Move a picture and delete a picture	0.03	0.03
AntennaPod (ap)	Podcast client	691K+	1.8.3	View 6 episode descriptions	0.03	0.03
OpenKeychain (ok)	Encryption	538K+	5.7.5	Encrypt and decrypt a file	0.02	0.01
Slide for Reddit (slide)	Online community	222K+	6.3	View 3 posts in 3 subreddits	0.08	0.02
Conversations (conv)	Instant messenger	127K+	2.8.9	Send 10 random messages	0.04	0.01
Download Navi (dn)	Download manager	75K+	1.4	Download a 100MB file	0.03	0.02
Wikimedia Commons (wc)	Image sharing	69K+	2.13	View 4 images	0.08	0.02
Etar Calendar (etar)	Calendar	39K+	1.0.26	Create 3 events	0.08	0.03
IPFS Lite (ipfs)	P2P Browser	4K+	2.5.4	View 5 Wikipedia articles	0.06	0.03
F-Droid (fdroid)	App store	N/A	1.8	View 3 app descriptions	0.06	0.03
F-Droid Build Status (build)	Continuous delivery	N/A	2.8.0	View 5 build logs	0.03	0.03
RadarWeather (rw)	Weather	N/A	4.4	View weather of 5 cities	0.04	0.03

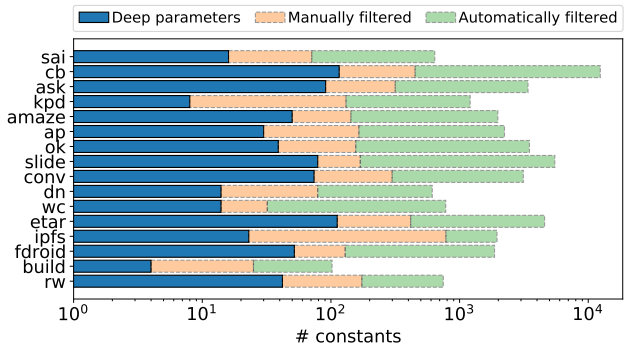


Fig. 7: Number of numeric parameters identified (blue), and of constants filtered out by manual (beige) after automatic (green) filtering. Other parameter types are omitted for space.

A. Energy Measurement

We run the experiments on two Pixel 2 phones, which are connected via USB to install app variants and accept UI automation commands. Since power meter readings tend to be less accurate when devices are connected [33], we use power models instead to measure energy consumption. We use well-established utilization-based power models [34], [35] to calculate CPU and GPU energy, and finite-state machine-based modeling [36]–[40] for WiFi. We calculate the power of each hardware component by collecting the relevant data (state and frequency information for CPU/GPU; transmission log for WiFi) using ftrace [41] and feeding them into the power models. As only the energy consumption of hardware components with power models can be calculated, we did not test apps that use specialized hardware components like hardware codecs or GPS.[4]

TABLE IV: Effectiveness of combining coverage- and heuristic-based filtering. Other apps are omitted for space.

	cb			kpd		
	Num	Bool	Enum	Num	Bool	Enum
No filtering	12402	844	269	1218	575	164
Coverage	1277	311	73	310	142	29
Heuristic	9404	236	112	606	184	113
Combined	451	115	31	131	71	13

VII. Results and Findings

A. Parameter Extraction

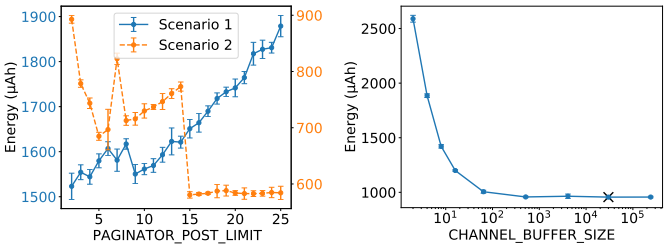
To speed up testing, we filter out unused and non-parameter constants by combining both automatic and manual filtering. Figure 7 shows the effect of each filtering method for numeric constants. Automatic filtering filters out 92.1% of the numeric constants. Manual filtering filters out another 6.2%, which further speeds up testing and leaves us on average 48 deep parameters per app. For Boolean constants the proportions are 90.3% automatic and 5.5% manual, leaving on average 40 deep parameters per app. For enumerator references, the proportions are 88.9% and 9.2% , and on average 15 are left.

To measure the effectiveness of each automatic filtering method, we turn them on and off individually and calculate the number of constants filtered out. Per Table IV, coverage-based filtering alone filters out on average 86.2% constants, while heuristic-based filtering alone filters out 31.1%. Combining them further improves the filtering efficiency to 94.8%, reducing subsequent manual filtering effort.

B. RQ3: Parameter-Induced Energy Inefficiency

To see whether energy-reducing deep parameters are common among apps, we mutate each deep parameter and measure the energy drain of each variant. After filtering the constants in the 16 apps, we get in total 764 numeric parameters,

[4]Prior work on GPS parameter tuning [9] used long-running experiments instead, and they estimated energy consumption by reading the battery level.



(a) The energy drain of `slide` with respect to the post fetching batch size under two usage scenarios.

(b) The energy drain of `cb` under different channel buffer sizes. "×" corresponds to the original (energy efficient) parameter value.

Fig. 8: The energy effects of two deep parameters. The error bars represent the standard deviation of 5 runs. Note the y-axes do not start from 0.

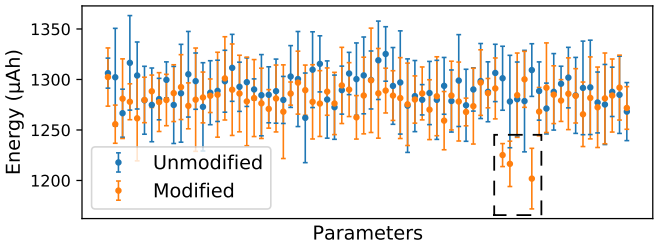


Fig. 9: For each of the 74 numeric parameters in `conv`, the energy consumption of the parameter value consuming the least energy (Modified) vs. the original parameter value (Unmodified). Error bars represent the standard deviation of 5 runs. Data points in dashed box have statistically significant reduction in energy drain.

TABLE V: Number of deep Parameters (P) tested for each app and parameter type, number of parameters that appear to Reduce the energy drain (R) during the test, and number of parameters manually Validated (V) to be energy-reducing.

	Numeric			Boolean			Enum		
	P	R	V	P	R	V	P	R	V
`sai`	16	2	0	22	0	0	10	0	0
`cb`	116	2	0	17	0	0	10	0	0
`ask`	91	0	0	35	0	0	17	0	0
`kpd`	8	0	0	37	0	0	5	0	0
`amaze`	50	0	0	25	0	0	15	0	0
`ap`	30	0	0	33	2	0	18	0	0
`ok`	39	0	0	55	4	0	14	0	0
`slide`	79	2	1	168	0	0	29	0	0
`conv`	74	3	0	58	1	0	24	0	0
`dn`	14	1	0	50	0	0	9	0	0
`wc`	14	1	0	1	1	0	7	0	0
`etar`	112	1	0	65	0	0	39	0	0
`ipfs`	23	1	1	18	2	0	3	0	0
`fdroid`	52	1	0	32	0	0	10	0	0
`build`	4	1	0	10	0	0	15	0	0
`rw`	42	1	0	21	0	0	8	2	0
Total	764	16	2	647	10	0	233	2	0

647 Boolean parameters, and 233 enumerator parameters. In testing all the parameters, we run the automated tests 15040 times (3008 parameter values with 5 runs each), which is 596 hours (25 days) of phone execution time.

Table V shows a summary of the test results. Out of the 1644 deep parameters tested, we observed reduced energy drain for 28 parameters. We then manually examined the 28 parameters, and found that only 2 numeric parameters really reduce the energy drain of the app without breaking app functionality.

Finding 3: Parameter-induced energy inefficiency is uncommon among apps. Out of the 1644 deep parameters from the 16 apps, only 2 reduce energy drain without breaking app functionality.

1) The True Positives: One energy-reducing parameter is identified in the Reddit client `slide`. While browsing the posts in a subreddit, the app fetches posts from the server. A batch of posts will be fetched each time, and each post will be processed immediately after being fetched. The energy used for processing is wasted if posts are fetched but not displayed.

The optimal batch size depends on the usage scenario. In Figure 8a, we measured the energy consumption with respect to the batch size for two different scenarios: view 3 posts in 3 subreddits (Scenario 1) and scroll 5 times in a subreddit feed (Scenario 2). The first scenario favors a smaller batch size, as it only loads the first screen for each subreddit. The second scenario favors a larger batch size as it loads more posts.

The other energy-reducing parameter, identified in the P2P browser `ipfs`, controls the ping interval to multiple peers. By reducing the ping frequency from once every second to once every 8 seconds, the app energy usage is reduced by 12.1% due to less frequent WiFi usage.

2) The False Positives: The other deep 26 parameters appear to reduce energy drain for three different reasons. The 3 numeric parameters in `conv` happened to correspond to the three reasons. Thus, we use these parameters to illustrate.

Figure 9 shows the test results of the 73 numeric parameters in `conv`. All standard deviations are within 4% (t_s) of the corresponding means. Most parameter values have energy consumption very close to the original values. The only three data points that have statistically significant energy difference are those in the dashed box (5.8%, 4.8%, 8.2%).

The rightmost data point represents the most common reason (12 of the 26 parameters): Even though we try to reduce false positives caused by the stochastic energy drain through measures like back-to-back testing, hypothesis testing, and stability threshold, sometimes the changes are too large and abrupt that the framework treats them as real energy reductions. The way to identify such cases is to rerun the tests and see if the energy reduction can be reproduced.

The data point in the middle corresponds to the second reason (13 of the 26 parameters): energy reduction at the cost of broken or degraded functionality. The `conv` parameter controls the refresh rate of various UI elements. By increasing the refresh interval from 500ms to 4000ms, the app energy consumption is reduced by 4.8%. However, when sending a text message, the message will take 4 seconds to appear in the conversation view. Other parameters have problems like

blanking out the app screen or disappearing all images.

The last and only one in its category is due to unexpected interaction between the app and our test automation script. When measuring energy consumption, we omit the initialization phase of the app and only measure the target test scenario. As we do not know exactly when all initializations are done, there may be some lingering initialization tasks after we have entered the test scenario, and their energy consumption will also be measured. This is not a problem as long as we enter the test scenario after the same delay. But the `conv` parameter delays entering the test scenario until all initializations have been done, leading to the apparent energy reduction.

C. RQ4: Parameters' Energy Effect

In the last section, we showed that deep parameter-induced energy inefficiency is uncommon among apps, and discussed the 2 energy-reducing parameters we discovered. In this section, we consider the "Why" question: *Why do deep parameters commonly not affect energy usage?*

To answer the question, we manually examine the 143 deep parameters in `cb`, and try to figure out their energy effects by understanding their semantics in the context of the source code and testing additional parameter values. We finally classified them into 3 categories based on their energy effect: having no energy effect, having limited energy effect, and having energy effect under extreme values.

1) Deep Parameters with No Energy Effect: 71 of the 143 parameters fall in this group and are further divided into two representative types. The first type of such parameters has binary effects. When the parameter value is in the valid range, the app works the same way regardless of the exact parameter value. On the other hand, the app breaks if the parameter value is in the invalid range. Lines 2-3 in Figure 1 shows two examples. Line 2 creates a buffer for version string parsing. App behavior is preserved when the buffer is big enough to hold the version string, but incorrect parsing occurs when the buffer is too small. In the second example, if the timeout of the socket is longer than the server's response latency, the socket communication works normally regardless of the exact timeout value (0 means indefinite timeout); the connection breaks if the timeout value is too small.

The other type of deep parameters without energy effect is due to limitations of our energy measurement methodology. We use power models to calculate the energy consumption of each hardware component. The change in energy consumption of a hardware component cannot be captured if the corresponding power model is missing. For example, the choice of colors can affect the energy consumption of OLED displays [42], but we omitted it since measuring the OLED energy consumption is expensive (we would have to record every frame).

2) Deep Parameters with Limited Energy Effects: 61 deep parameters fall in this category. Each parameter is typically attached to a certain component of the app. Thus, the energy effect of the parameter depends on both the total energy consumption of the component and the importance of the parameter in the component. The logging component typically consumes a limited amount of energy for most apps; the parameters controlling logging levels will thus have limited energy effect. On the other hand, although UI is energy expensive, the energy consumption of UI rendering mainly depends on the structure of the UI element tree, instead of the precise positions and sizes of the individual UI elements, and thus these UI parameters also have a limited energy effects.

3) Deep Parameters Having Effects under Extreme Values: To see how energy drain can be affected by extreme parameter values, we will first look at an example. Figure 8b shows the energy drain of `cb` under varying `CHANNEL_BUFFER_SIZE`, which controls the size of the `stdout` and `stderr` buffers attached to the terminal. Extremely small buffers divide terminal outputs into small chunks, and processing them one by one adds overhead. Such extreme values only occupy a tiny fraction of the valid value range, thus are not captured by our framework. However, a developer is also unlikely to pick such extreme values if she understood the meaning of the parameter.

Apart from buffer sizes, making the font size of the terminal extremely small also increases energy consumption drastically. In total 9 parameters in `cb` are of this kind. Such parameters also exist in other apps. For example, the UI update frequency parameter discussed in the last section also only exhibits an energy effect when the update interval is extremely long. Similarly, a developer will not choose such extreme values if she considered the semantics of the parameters. Basically, developers can typically avoid such extreme values based on their domain knowledge.

> **Finding 4**: Most deep parameters either have no energy effect, limited energy effect, or only have energy effect under extreme values. We expect developers would typically avoid such extreme values based on their domain knowledge.

VIII. Discussion and Future Work

a) Potential impact factors to energy consumption: Our work is the first systematic attempt to understand the energy impact of deep parameters in mobile apps. *Across 16 apps, we found that mobile deep parameters did not have a significant impact on app energy.* We conjecture three possible explanations. First, it may be that the app's design — the software architecture and design patterns [43] — has a dominant effect on the app's energy usage [44]–[46]. Second, it may be that our constraint was too strong; mobile apps may have to sacrifice user experience or remove features to conserve energy. Third, while individual parameters cannot move the needle, tuning them in combination might have a bigger impact [47], [48]. Each of these possibilities is a direction for further study.

b) Automatic support for tuning deep parameters: Most parameter tuning systems work only with the "formal" parameters exposed by developers in a central repository (cf. §X). This design assumes that developers have identified and centralized their parameters. However, when exposing parameters by hand, it is difficult to anticipate the needs of

future use cases. Wang *et al.* [49] discussed difficulties in tuning database systems because developers had hard-coded deep parameters instead of exposing them for tuning.

One strength of our deep parameter-identification framework (Figure 3) is that we *automatically* identify deep parameters. In the future, we plan to apply our deep parameter search framework to other classes of software (*e.g.,* database systems) and help discover those important deep parameters. In these contexts, we will develop a unified parameter tuning approach that merges formal and deep parameters.

c) Large-scale energy measurement: Current mobile phone energy measurement methods (both power monitors and power models) rely on real phones, making energy measurement unscalable. In our experiments, it took on average 22 minutes to test each parameter, which means roughly 1.5 days per app. Accurate energy measurement in virtualized environments will enable larger-scale experiments on energy optimization. Accurate emulation of the hardware states and frequencies for power modeling is one possible direction.

IX. Threats to Validity

a) Internal Validity: There are several threats to internal validity. Survey: Although we refined our survey instrument through pilot studies, it has not been validated [50]. We assume our respondents replied honestly. Energy experiments: Energy changes might be due to factors other than the mutated parameter. Such factors include changes in timing, network conditions, and external service behaviors. This threat is mitigated by using automated testing and repeated trials for each deep parameter.

b) External Validity: Our findings may not generalize to different classes of software [51]. Within Android apps, there may be differences between the 16 open-source apps we investigated vs. (1) commercial apps, and (2) apps that are deliberately designed to be energy-efficient. For some insight on this threat, most of our survey respondents develop Android apps commercially. They indicated that their parameters and parameterizations were not designed for energy efficiency.

c) Construct Validity: We define a deep parameter as a constant, and our experimental design preserves each parameterization throughout the lifetime of each measurement. Similarly, we tune deep parameters on a (static) per-class basis rather than a (dynamic) per-instance basis. This definition is generally consistent with how these constructs are defined in the apps we studied. However, a dynamic notion of deep parameters might affect our results; for example, the energy-optimal parameter choice has been shown to vary dynamically for GPS parameters in Android apps [9].

X. Related Work

a) Configuration tuning: Tuning software configuration parameters for better performance is a common practice for many classes of software. Recently, many automated configuration tuning systems are proposed, either for arbitrary configurable systems [6], [49], [52]–[55], or for specific application types [7], [56]. Tuning is conducted either offline, optimizing parameter values for a fixed workload and environment [57], [58], or online, dynamically reconfiguring the target system to adapt to changes [59], [60].

Most such works are focused on systems software, *e.g.,* file systems and databases. Only Bokhari *et al.* [10], [11] and Canino *et al.* [9] have considered deep parameters in Android apps. They focus on deep parameters in specific app components. Our work is the first step to understanding the energy impact of deep parameters in general Android apps; we study the energy impact of single parameters, and leave combinatorial tuning to future work.

b) Performance modeling: Many works [61]–[65] focus on building a performance model for a certain application and workload. A performance model is a mathematical function where the domain is the configuration parameters and the codomain is the performance. Performance optimization or other tasks can be further performed based on the performance model. These systems mostly rely on sampling, and they generate a better performance model by sampling efficiently. These works rely on explicitly exposed parameters. We consider instead a program's deep parameters.

c) Energy impacts of design patterns and refactoring: Researchers have studied the energy impacts of design patterns across software domains. Sahin *et al.* [66] compared the power profile of data center software using design patterns against those not using. Pinto *et al.* [67] focus on the energy consumption of Java thread management constructs.

Refactoring energy-greedy code patterns can also reduce the energy drain of mobile apps. Carette *et al.* [44] design a framework that automatically refactors Android code smells and observe reduced energy drain after correcting those smells. Cruz and Abreu [45] study refactorings for energy efficiency in the wild by mining source code commits, issues, and pull requests. Couto *et al.* [46] further study the impacts of refactoring on energy consumption by applying combinations of refactorings to a large set of Android apps. Their guidelines help developers reduce energy drain through refactoring. Our work considers energy improvement through parameterization instead of refactoring.

d) Mobile app energy testing: Discovering energy inefficiency through testing is an ongoing research topic. Ding and Hu [35] uncover the potential energy inefficiency during the rendering process. Jindal and Hu [68] discover energy-inefficient components by comparing them with other apps with similar functionalities. Jabbarvand *et al.* [69] enhances UI automated testing techniques to cover energy-heavy APIs, but lack proper oracles for unknown energy defects. As a first step towards automated energy test oracle construction, their subsequent work [70] employed deep learning to determine energy efficiency based on lifecycle and hardware states. Li *et al.* [71] classified mobile app energy issues into 6 categories and proposed different methods to detect the energy issues of each category. Our work focuses on constructs at a finer granularity, and is complementary to those works.

XI. Conclusion

We studied the energy impact of mobile deep parameters. We used a developer survey to understand the perceived impact, and a systematic experiment to understand the actual impact. Our survey showed that many app developers are uncertain about and ignore the energy impact of deep parameters. Our experiment and analysis with 16 apps showed that single-parameter-induced energy inefficiency is uncommon. However, in order to more fully explore energy-feature tradeoffs, in future work we plan to explore energy optimization opportunities through tuning combinations of deep parameters as well as through non-functionality-preserving parameterizations. For now, it appears that mobile app developers can ignore the energy impact when choosing deep parameter values — they will not substantially degrade their app's energy performance.

References

[1] C. Wilke, S. Richly, S. Götz, C. Piechnick, and U. Aßmann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, 2013, pp. 134–141.

[2] A. K. Pramanik, "The Technology That's Making a Difference in the Developing World," https://www.usglc.org/blog/the-technology-thats-making-a-difference-in-the-developing-world/, 2017, accessed: 2021-04-20.

[3] P. Research, "Mobile Connectivity in Emerging Economies," https://www.pewresearch.org/internet/2019/03/07/mobile-connectivity-in-emerging-economies/, accessed: 2021-04-20.

[4] (2021, Feb.) Optimize for battery life. [Online]. Available: https://developer.android.com/topic/performance/power

[5] Energy efficiency and the user experience. [Online]. Available: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/index.html

[6] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 338–350.

[7] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1009–1024.

[8] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2118–2130, Aug. 2019.

[9] A. Canino, Y. D. Liu, and H. Masuhara, "Stochastic energy optimization for mobile gps applications," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 703–713.

[10] M. A. Bokhari, B. R. Bruce, B. Alexander, and M. Wagner, "Deep parameter optimisation on android smartphones for energy minimisation: A tale of woe and a proof-of-concept," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1501–1508.

[11] M. A. Bokhari, B. Alexander, and M. Wagner, "In-vivo and offline optimisation of energy use in the presence of small energy signals: A case study on a popular android library," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ser. MobiQuitous '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 207–215.

[12] R. B. Johnson and A. J. Onwuegbuzie, "Mixed methods research: A research paradigm whose time has come," *Educational researcher*, vol. 33, no. 7, pp. 14–26, 2004.

[13] Command line options. [Online]. Available: https://x265.readthedocs.io/en/master/cli.html

[14] Application programming interface. [Online]. Available: https://x265.readthedocs.io/en/master/api.html

[15] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, "Using probabilistic reasoning to automate software tuning," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '04/Performance '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 404–405.

[16] Questionnaire design. [Online]. Available: https://www.pewresearch.org/methods/u-s-survey-research/questionnaire-design/

[17] Survey design: How to design a survey that people will love to answer? [Online]. Available: https://www.questionpro.com/features/survey-design/

[18] Google play. [Online]. Available: https://play.google.com/store/apps

[19] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 396–409.

[20] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 472–483.

[21] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 115–127.

[22] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang, and H. Duan, "Textexerciser: Feedback-driven text input exercising for android applications," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1071–1087.

[23] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of android applications by constructing activity launching contexts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 457–468.

[24] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 481–492.

[25] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: Generating high-quality test inputs for android apps via use case combinations," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 469–480.

[26] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 153–164.

[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269.

[28] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 393–407.

[29] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 209–222, Jan 2020.

[30] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[31] Appium. [Online]. Available: http://appium.io/

[32] Jacoco java code coverage library. [Online]. Available: https://www.eclemma.org/jacoco/

[33] R. Jabbarvand and S. Malek, "Mudroid: An energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 208–219.

[34] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone background activities in the wild: Origin, energy drain, and optimization," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 40–52.

[35] N. Ding and Y. C. Hu, "Gfxdoctor: A holistic graphics energy profiler for mobile devices," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 359–373.

[36] A. Nika, Y. Zhu, N. Ding, A. Jindal, Y. C. Hu, X. Zhou, B. Y. Zhao, and H. Zheng, "Energy and performance of smartphone radio bundling in outdoor environments," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015, p. 809–819.

[37] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4g lte networks," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 225–238.

[38] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 43–55.

[39] L. Sun, R. K. Sheshadri, W. Zheng, and D. Koutsonikolas, "Modeling wifi active power/energy consumption in smartphones," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 41–51.

[40] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: A cross-layer approach," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 321–334.

[41] (2020) Using ftrace. [Online]. Available: https://source.android.com/devices/tech/debug/ftrace

[42] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for oled smartphones," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 527–538.

[43] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[44] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 115–126.

[45] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2209–2235, 2019.

[46] M. Couto, J. Saraiva, and J. P. Fernandes, "Energy refactorings for android in the large and in the wild," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 217–228.

[47] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin, "Exploring feature interactions in the wild: The new feature-interaction challenge," in *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, ser. FOSD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1–8.

[48] S. Kolesnikov, N. Siegmund, C. Kästner, and S. Apel, "On the relation of control-flow and performance feature interactions: a case study," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2410–2437, Aug. 2019.

[49] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 154–168.

[50] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 63–92.

[51] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, pp. 466–476.

[52] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the configuration for the heart of the os: On the practicality of operating system kernel debloating," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 1, May 2020.

[53] L. Bao, X. Liu, F. Wang, and B. Fang, "Actgan: Automatic configuration tuning for software systems with generative adversarial networks," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 465–476.

[54] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1375–1382.

[55] M. Kambadur and M. A. Kim, "An experimental survey of energy management across the stack," *SIGPLAN Not.*, vol. 49, no. 10, p. 329–344, Oct. 2014.

[56] Z. Cao, G. Kuenning, and E. Zadok, "Carver: Finding important parameters for storage system tuning," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 43–57.

[57] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok, "Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 893–907.

[58] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[59] A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi, "SOPHIA: Online reconfiguration of clustered nosql databases for time-varying workloads," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 223–240.

[60] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E. Long, "Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017.

[61] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 284–294.

[62] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 71–82.

[63] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 1084–1094.

[64] D. Rogora, A. Carzaniga, A. Diwan, M. Hauswirth, and R. Soulé, "Analyzing system performance with probabilistic performance annotations," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[65] M. Velez, P. Jamshidi, F. Sattler, N. Siegmund, S. Apel, and C. Kästner, "ConfigCrusher: towards white-box performance analysis for configurable systems," *Automated Software Engineering*, Aug. 2020.

[66] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, 2012, pp. 55–61.

[67] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," *SIGPLAN Not.*, vol. 49, no. 10, p. 345–360, Oct. 2014.

[68] "Differential energy profiling: Energy optimization via diffing similar apps," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018.

[69] R. Jabbarvand, J. Lin, and S. Malek, "Search-based energy testing of android," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1119–1130.

[70] R. Jabbarvand, F. Mehralian, and S. Malek, "Automated construction of energy test oracles for android," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 927–938.

[71] X. Li, Y. Yang, Y. Liu, J. P. Gallagher, and K. Wu, "Detecting and diagnosing energy issues for mobile applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 115–127.